С Е Р И Я

КЛАССИКА COMPUTER SCIENCE



Robert Lafore

Data Structures & Algorithms in Java

Second Edition



800 East 96th Street, Indianapolis, Indiana 46240



РОБЕРТ ЛАФОРЕ

СТРУКТУРЫ ДАННЫХ И АЛГОРИТМЫ JAVA

ВТОРОЕ ИЗДАНИЕ



Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2013

ББК 32.973.2-018
УДК 004.422.63
Л29

Лафоре Р.
Л29 Структуры данных и алгоритмы в Java. Классика Computers Science. 2-е изд. — СПб.: Питер, 2013. — 704 с.: ил. — (Серия «Классика computer science»).

ISBN 978-5-496-00740-5

Второе издание одной из самых авторитетных книг по программированию посвящено использованию структур данных и алгоритмов. Алгоритмы — это основа программирования, определяющая, каким образом разрабатываемое программное обеспечение будет использовать структуры данных. На четких и простых программных примерах автор объясняет эту сложную тему, предлагая читателям написать собственные программы и на практике усвоить полученные знания. Рассматриваемые примеры написаны на языке Java, хотя для усвоения материала читателю не обязательно хорошо знать его — достаточно владеть любым языком программирования, например C++. Первая часть книги представляет собой введение в алгоритмизацию и структуры данных, а также содержит изложение основ объектно-ориентированного программирования. Следующие части посвящены различным алгоритмам и структурам данных, рассматриваемым от простого к сложному: сортировка, абстрактные типы данных, связанные списки, рекурсия, древовидные структуры данных, хеширование, пирамиды, графы. Приводятся рекомендации по использованию алгоритмов и выбору той или иной структуры данных в зависимости от поставленной задачи.

12+ (Для детей старше 12 лет. В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018
УДК 004.422.63

ISBN 978-0672324536 англ.
ISBN 978-5-496-00740-5

Оглавление

Посвящаю эту книгу моим читателям, которые поддерживали меня все эти годы не только покупкой моих книг, но и полезными предложениями и добрым словом. Спасибо вам всем

Введение

В этом разделе вы узнаете:

- ✦ Что нового появилось во втором издании книги.
- ✦ О чем эта книга.
- ✦ Чем она отличается от других книг.
- ✦ Кому она может пригодиться.
- ✦ Что необходимо знать перед чтением.
- ✦ Какие программы и оборудование могут понадобиться читателю.
- ✦ Как организован материал книги.

Второе издание

Второе издание книги было доработано для того, чтобы читателям было проще усваивать материал, а преподавателям — использовать его на занятиях по программированию. Помимо ряда дополнительных тем, в конце каждой главы появились контрольные вопросы, эксперименты и программные проекты для самостоятельной работы.

Дополнительные темы

В книге рассматривается целый ряд новых интересных тем, которые можно использоваться для создания программных проектов. Вот лишь несколько примеров:

- ✦ Обход в глубину и игровое программирование.
- ✦ Задача Иосифа Флавия.
- ✦ Коды Хаффмана и сжатие данных.
- ✦ Задача коммивояжера.
- ✦ Гамильтоновы циклы.
- ✦ Задача обхода доски ходом шахматного коня.
- ✦ Алгоритм Флойда.
- ✦ Алгоритм Уоршелла.
- ✦ Деревья 2-3.

- Задача о рюкзаке.
- Генерирование перестановок *K* объектов из *N*.
- Применение свертки при хешировании.
- Поразрядная сортировка.

Вопросы

В конце каждой главы приводится список вопросов по ключевым положениям этой главы. Ответы на них приведены в приложении В. Вопросы предназначены для самоконтроля — по ним читатели смогут проверить, насколько глубоко усвоен материал.

Упражнения

Мы также предлагаем несколько полезных упражнений для читателя. Одни упражнения связаны с выполнением каких-либо операций с приложениями Workshop или примерами программ для исследования особенностей алгоритмов, другие выполняются с карандашом и бумагой, а то и вовсе относятся к категории «умозрительных экспериментов».

Программные проекты

И самое важное: в конце каждой главы читателю предлагается несколько (обычно пять) программных проектов. Простейшие проекты представляют собой простые вариации на тему программ-примеров, а самые сложные — реализации тем, которые упоминались в тексте без приведения программного кода. Решения в книге не приводятся.

О чем эта книга

Книга посвящена использованию структур данных и алгоритмов в программировании. Структуры данных определяют способ организации данных в памяти компьютера (или на диске). Алгоритмы обеспечивают выполнение различных операций с этими структурами.

Структуры данных и алгоритмы используются почти во всех компьютерных программах, включая самые простые. Для примера возьмем программу для печати адресных наклеек. Такая программа может использовать массив с адресами; содержимое массива перебирается в простом цикле `for`, и каждый адрес выводится на печать.

Массив в этом примере является структурой данных, а цикл `for`, обеспечивающий последовательный доступ к элементам, выполняет простой алгоритм. Для

простых программ с небольшими объемами данных такого простого решения может оказаться достаточно. Но для программ, обрабатывающих хотя бы умеренные объемы данных или решающих хоть сколько-нибудь нетривиальные задачи, требуются более сложные методы. Простого знания синтаксиса языка программирования, будь то Java или C++, недостаточно.

В этой книге рассказывается о том, что необходимо знать после изучения языка программирования. Изложенный материал обычно преподается в институтах и университетах на второй год преподавания информатики, после того как студент освоит азы программирования.

Чем эта книга отличается от других

О структурах данных и алгоритмах написаны десятки книг. Чем эта книга отличается от них? Три основных отличия:

- Работая над книгой, мы прежде всего стремились к тому, чтобы материал излагался понятно и доступно.
- Демонстрационные приложения Workshop наглядно поясняют рассматриваемые темы, шаг за шагом показывая, как работают структуры данных и алгоритмы.
- Примеры кода написаны на языке Java, более понятном, чем C, C++ или Pascal (языки, традиционно использовавшиеся для написания примеров в учебниках по программированию).

Давайте рассмотрим каждое отличие более подробно.

Доступность

Типичный учебник по программированию содержит теорию, математические формулы и заумные примеры кода. В этой книге, напротив, основное внимание уделяется простому объяснению методов решения практических задач. Мы избегаем сложных обоснований и математических выкладок. Текст поясняется многочисленными иллюстрациями.

Многие книги о структурах данных и алгоритмах включают материал, относящийся к программотехнике — дисциплине, направленной на проектирование и реализацию крупных и сложных программных проектов. Однако мы полагаем, что структуры данных и алгоритмы сложны и без привлечения дополнительных дисциплин, поэтому мы намеренно отказались от рассмотрения программотехнических вопросов в книге. (О связи структур данных и алгоритмов с программотехникой рассказано в главе 1, «Общие сведения»).

Конечно, мы используем объектно-ориентированный подход и рассматриваем различные аспекты объектно-ориентированной архитектуры, включая мини-курс ООП в главе 1. И все же основное внимание уделяется самим структурам данных и алгоритмам.

Приложения Workshop

С сайтов издательств «Питер» и Sams можно загрузить демонстрационные приложения Workshop для обсуждаемых тем. Приложения оформлены в виде апплетов Java, работающих в большинстве современных браузеров. (За дополнительной информацией обращайтесь к приложению А.) Приложения Workshop строят графические схемы, которые показывают, как работает алгоритм в режиме «замедленной съемки». Например, в одном из приложений при каждом нажатии кнопки на гистограмме выполняется очередной шаг процесса сортировки столбцов по возрастанию. Также выводятся значения переменных, задействованных в алгоритме сортировки, чтобы при выполнении алгоритма было точно видно, как работает программный код. Текстовые сообщения поясняют, что происходит в данный момент.

Другое приложение моделирует двоичное дерево. При помощи стрелок, перемещающихся по дереву вверх-вниз, пользователь следит за тем, что происходит при вставке или удалении узла из дерева. В книге используется более 20 приложений Workshop — не менее одного для каждой из основных тем.

Приложения Workshop объясняют, как в действительности устроена структура данных или как работает алгоритм, намного эффективнее любого текстового описания. Конечно, текстовые описания тоже присутствуют. Именно сочетание приложений Workshop, понятного текста и иллюстраций упрощает восприятие материала.

Приложения Workshop являются автономными графическими программами. Их можно использовать как обучающие средства, дополняющие материал книги. Не путайте приложения Workshop с примерами кода, приводимыми в тексте книги, — о них будет рассказано ниже.

Примеры кода Java

Код на языке Java более понятен (и проще пишется), чем код на языках вроде C или C++. В основном это связано с тем, что в Java не используются указатели. Возможно, кому-то покажется удивительным, что создание сложных структур данных и алгоритмов может обойтись без указателей. На самом деле отказ от указателей не только упрощает написание и понимание кода, но и повышает его надежность и устойчивость к ошибкам.

Так как Java является современным объектно-ориентированным языком, мы можем использовать объектно-ориентированный подход в своих примерах кода. Это очень важно, потому что объектно-ориентированное программирование (ООП) во многих отношениях превосходит старый процедурный подход и быстро вытесняет его в серьезных проектах. Если вы еще не знакомы с методологией ООП, не огорчайтесь — разобраться в ней несложно, особенно при отсутствии указателей, как в языке Java. Основы ООП излагаются в главе 1.

Для кого написана эта книга

Книга может использоваться как изложение учебного курса «Структуры данных и алгоритмы», который обычно читается на второй год преподавания информатики. Впрочем, она также пригодится профессиональным программистам — и вообще всем, кто захочет сделать следующий шаг после изучения языка программирования. Ввиду доступности материала книга также может рассматриваться как дополнительный источник информации для более академичного и формального курса.

Что необходимо знать читателю

От читателя требуется только одно — владение каким-либо языком программирования.

Хотя примеры кода написаны на Java, вам не обязательно знать Java, чтобы разобраться в происходящем. Язык Java понять несложно, а мы постарались ограничиться общим синтаксисом, по возможности избегая экзотических или существующих только в Java конструкций.

Конечно, если читатель уже знаком с Java, это пойдет ему только на пользу. Также пригодится знание C++, так как синтаксис Java строился на базе C++. Для наших программ различия между этими языками несущественны (если не считать столь полезного исчезновения указателей); они будут кратко рассмотрены в главе 1.

Необходимые программы

Для запуска приложений Workshop вам понадобится браузер — например, Microsoft Internet Explorer или Netscape Communicator. Также может пригодиться программа для просмотра апплетов. Такие программы входят в комплект многих систем программирования на Java, в том числе и бесплатного пакета от Sun Microsystems (см. приложение А).

Для запуска примеров программ понадобится окно командной строки в Microsoft Windows или аналогичная консольная среда.

Если вы захотите изменить исходный код примеров или написать собственную программу, вам понадобится система разработки на языке Java. Существует несколько коммерческих систем такого рода; кроме того, отличную систему базового уровня можно загрузить с сайта Sun Microsystems (см. приложение А).

Как организован материал книги

Информация в этом разделе предназначена для преподавателей и читателей, которые хотят получить представление о содержимом книги. Предполагается, что читатель уже знаком с основными понятиями и терминами из области структур данных и алгоритмов.

Первые две главы содержат простое и доступное введение в область структур данных и алгоритмов.

Глава 1, «Общие сведения», содержит общий обзор обсуждаемых тем. Кроме того, в ней вводятся некоторые термины, используемые в книге. Для читателей, незнакомых с объектно-ориентированным программированием, приводится сводка тех аспектов ООП, которые могут понадобиться при изучении материала книги. Для читателей, знающих C++, но не владеющих Java, перечислены важнейшие различия между двумя языками.

Глава 2, «Массивы», посвящена массивам. Впрочем, в ней имеются два подраздела: об использовании классов для инкапсуляции структур данных и об интерфейсе классов. В главе рассмотрены операции поиска, вставки и удаления в массивах и упорядоченных массивах, объясняется суть линейного и двоичного поиска. Приложения Workshop демонстрируют работу этих алгоритмов для неупорядоченных и упорядоченных массивов.

В главе 3, «Простая сортировка», представлены три простых (но медленных) алгоритма сортировки: пузырьковая сортировка, сортировка методом выбора и сортировка методом вставки. Каждый алгоритм продемонстрирован соответствующим приложением Workshop.

В главе 4, «Стеки и очереди», рассматриваются три структуры данных, которые можно условно отнести к абстрактным типам данных (ADT): стек, очередь и приоритетная очередь. Позднее в книге мы еще вернемся к этим структурам при изучении различных алгоритмов. Каждая структура продемонстрирована соответствующим приложением Workshop. Также в главе обсуждается концепция абстрактных типов данных.

Глава 5, «Связанные списки», знакомит читателя со связанными списками, включая двусвязные и двусторонние списки. Объясняется роль ссылок в языке Java как «указателей без проблем». Приложение Workshop показывает, как выполняются операции вставки, поиска и удаления в связанных списках.

Глава 6, «Рекурсия», полностью посвящена рекурсии — это одна из немногих глав, в которой не рассматривается никакая конкретная структура данных. Приводится несколько примеров рекурсии, в числе которых головоломка «Ханойская башня» и сортировка слиянием. Эти примеры также поясняются приложениями Workshop.

В главе 7, «Нетривиальная сортировка», мы переходим к более совершенным методам: сортировке Шелла и быстрой сортировке. Приложения Workshop демонстрируют сортировку Шелла, разбиение (основу алгоритма быстрой сортировки) и две вариации быстрой сортировки.

С главы 8, «Двоичные деревья», начинается изучение темы деревьев. В этой главе рассмотрена простейшая древовидная структура данных: несбалансированное дерево двоичного поиска. Приложение Workshop демонстрирует вставку, удаление и обход таких деревьев.

В главе 9, «Красно-черные деревья», рассматривается одна из самых эффективных разновидностей сбалансированных деревьев — красно-черные деревья. Приложение Workshop демонстрирует операции поворота и переключения цветов, необходимые для балансировки дерева.

В главе 10, «Деревья 2-3-4», описаны деревья 2-3-4 как пример многопутевых деревьев. Приложение Workshop показывает, как работают деревья 2-3-4. Также в главе рассматриваются деревья 2-3 и отношение деревьев 2-3-4 с B-деревьями, используемыми для внешнего хранения данных (в дисковых файлах).

Глава 11, «Хеш-таблицы», переходит к новой области — хешированию. Приложения Workshop демонстрируют разные методы хеширования: линейное и квадратичное пробирование, двойное хеширование и метод цепочек. Рассматривается возможность применения хеширования для организации внешнего хранения файлов.

В главе 12, «Пирамиды», рассматривается пирамида — специализированное дерево, используемое для эффективной реализации приоритетных очередей.

Глава 13, «Графы», и глава 14, «Взвешенные графы», посвящены графам. В первой рассматриваются невзвешенные графы и простые алгоритмы поиска, а во второй — взвешенные графы и более сложные алгоритмы, применяемые для построения минимальных охватывающих деревьев и кратчайших путей.

В главе 15, «Рекомендации по использованию», приводится сводка структур данных из предыдущих глав. Особое внимание уделяется выбору структуры для конкретной ситуации.

Приложение А, «Приложения Workshop и примеры программ», содержит подробные инструкции об использовании этих двух видов программных продуктов. Кроме того, в нем рассказано, как использовать пакет SDK от Sun Microsystems для изменения примеров и написания собственных программ, запуска приложений Workshop и примеров.

В приложении Б, «Литература», перечислены некоторые книги для дальнейшего изучения структур данных и других сопутствующих тем.

В приложении В, «Ответы на вопросы», содержатся ответы на вопросы, приведенные в конце глав книги.

Вперед!

Хочется верить, что нам удалось по возможности избавить читателя от многих проблем. В идеале процесс обучения должен приносить удовольствие. Сообщите, удалось ли нам достичь этой цели, и если нет, что, по вашему мнению, нуждается в совершенствовании.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Приложения и примеры можно найти на странице этой книги на веб-сайте издательства «Питер» (www.piter.com) и на сайте Sams по адресу www.samspublishing.com.

Мы будем рады узнать ваше мнение!

На веб-сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

Глава 1
Общие сведения

В самом начале книги у читателей могут возникнуть некоторые вопросы:

- Что такое «структуры данных и алгоритмы»?
- Какую пользу принесет мне их изучение?
- Почему бы не использовать для работы с данными массивы и циклы `for`?
- Когда следует применять те знания, которые я найду в книге?

В этой главе я постараюсь ответить на эти вопросы. Также в ней будут представлены некоторые термины, которые необходимо знать, и подготовлена почва для изложения материала следующих, более подробных глав.

Для тех читателей, которые еще не сталкивались с объектно-ориентированными языками, будут представлены основные положения ООП. Наконец, для программистов C++, не владеющих языком Java, приводится сравнительный обзор этих языков.

Зачем нужны структуры данных и алгоритмы?

Эта книга посвящена структурам данных и алгоритмам. *Структурой данных* называется способ организации данных в памяти компьютера (или на диске). Примерами структур данных являются массивы, связанные списки, стеки, двоичные деревья, хеш-таблицы и т. д. *Алгоритмы* обеспечивают выполнение различных операций с этими структурами — например, поиск определенного элемента данных или сортировку данных.

Какие задачи помогает решить хорошее знание структур данных и алгоритмов? Ситуации, в которых они могут пригодиться, можно приблизительно разделить на три категории:

- Хранение реальных данных.
- Инструментарий программиста.
- Моделирование.

Данная классификация не является жесткой и формальной, но она дает представление о полезности материала. Рассмотрим эти три категории более подробно.

Хранение реальных данных

Многие структуры и приемы, рассматриваемые в книге, направлены на решение задач хранения реальных данных. Под реальными данными подразумеваются данные, описывающие физические сущности, внешние по отношению к компьютеру. Например, запись в базе данных персонала описывает конкретного человека; запись в складской базе данных описывает деталь для автомобиля или бакалейный товар, а запись в базе данных финансовых операций — выписку счета за электричество.

«Физическим» примером хранения реальных данных служит картотека. Карточки могут использоваться для разных целей. Если записать на них имена, адреса и телефонные номера, мы получим адресную книгу. Если же на карточках записаны названия, местонахождение и стоимость предметов, то результатом будет инвентарная опись.

Конечно, картотеку вряд ли можно признать современным решением. Практически все, что когда-то делалось с помощью карточек, теперь может быть сделано на компьютере. Допустим, вы хотите преобразовать свою старую картотеку в компьютерную программу. Вероятно, в процессе преобразования придется найти ответы на следующие вопросы:

- Как данные будут храниться в памяти компьютера?
- Подойдет ли ваше решение для сотни карточек? Для тысячи? Миллиона?
- Позволяет ли ваше решение быстро добавлять новые и удалять старые карточки?
- Возможен ли быстрый поиск заданной карточки?
- Допустим, вы хотите отсортировать карточки в алфавитном порядке. Как будет происходить сортировка?

В этой книге рассматриваются структуры данных, процесс использования которых имеет нечто общее с традиционной картотекой.

Конечно, большинство программ сложнее картотек. Представьте себе базу данных с информацией о водительских правах или систему заказа авиабилетов с информацией о пассажирах и рейсах. В таких системах могут использоваться разные структуры данных. При проектировании сложных систем применяются методы программотехники, о которых мы расскажем ближе к концу главы.

Инструментарий программиста

Не все структуры данных используются для хранения реальных данных. Как правило, реальные данные более или менее непосредственно доступны для пользователя программы. Но существуют и другие структуры данных, с которыми должен работать не пользователь, а сама программа. Программист использует такие структуры как вспомогательные инструменты для выполнения других операций. В частности, стеки, очереди и приоритетные очереди часто используются подобным образом. В дальнейшем мы увидим и другие примеры вспомогательных структур данных.

Моделирование

Некоторые структуры данных непосредственно моделируют ситуации из реального мира. Важнейшей структурой данных этого типа является граф. Графы могут использоваться для представления авиарейсов между городами, соединений в электрической цепи или задач в проекте. Графы более подробно рассматриваются в главах 13, «Графы», и 14, «Взвешенные графы». Другие структуры данных, в том числе стеки и очереди, тоже могут использоваться в моделировании. Скажем, очередь может моделировать клиентов, ожидающих обслуживания в банке, или машины, ожидающие въезда на платное шоссе.

Обзор структур данных

Другой подход к изучению структур данных основан на анализе их сильных и слабых сторон. В этом разделе приведен обзор (в форме таблицы) важнейших структур данных, которые будут рассматриваться в книге. Считайте, что мы с высоты птичьего полета рассматриваем ландшафт, который будет подробно изучаться позднее, так что не беспокойтесь, если какие-то термины будут вам незнакомы. В таблице 1.1 перечислены достоинства и недостатки различных структур данных, описанных в книге.

Таблица 1.1. Характеристики структур данных

Структура данных	Достоинства	Недостатки
Массив	Быстрая вставка, очень быстрый доступ (если известен индекс)	Медленный поиск, медленное удаление, фиксированный размер
Упорядоченный массив	Поиск выполняется быстрее, чем в обычном массиве	Медленная вставка и удаление, фиксированный размер
Стек	Доступ в порядке LIFO («последним пришел, первым вышел»)	Медленный доступ к другим элементам
Очередь	Доступ в порядке FIFO («первым пришел, первым вышел»»)	Медленный доступ к другим элементам
Связанный список	Быстрая вставка, быстрое удаление	Медленный поиск
Двоичное дерево	Быстрый поиск, вставка, удаление (если дерево остается сбалансированным)	Сложный алгоритм удаления
Красно-черное дерево	Быстрый поиск, вставка, удаление. Дерево всегда сбалансировано	Сложность
Дерево 2-3-4	Быстрый поиск, вставка, удаление. Дерево всегда сбалансировано. Аналогичные деревья хорошо подходят для хранения данных на диске	Сложность

продолжение ⇨

Таблица 1.1 (*продолжение*)

Структура данных	Достоинства	Недостатки
Хеш-таблица	Очень быстрый доступ (если известен ключ)	Медленное удаление, медленный доступ (если ключ неизвестен), неэффективное использование памяти
Куча	Быстрая вставка, удаление, доступ к наибольшему элементу	Медленный доступ к другим элементам
Граф	Моделирование реальных ситуаций	Некоторые алгоритмы медленны и сложны

Все структуры данных из табл. 1.1, за исключением массивов, могут рассматриваться как абстрактные типы данных (ADT). О том, что это такое, рассказано в главе 5, «Связанные списки».

Алгоритмы

Многие алгоритмы, описанные в книге, предназначены для работы с конкретными структурами данных. Для большинства структур данных необходимо знать, как выполняются операции:

- Вставки нового элемента данных.
- Поиска заданного элемента.
- Удаления заданного элемента.

В некоторых случаях также необходимо знать, как выполняется *перебор* всех элементов структуры данных, то есть последовательного получения каждого элемента для выполнения некоторой операции (например, вывода).

Другую важную категорию алгоритмов составляют алгоритмы *сортировки*. Существует много различных способов сортировки данных; им посвящены глава 3, «Простая сортировка», и глава 7, «Нетривиальная сортировка».

В некоторых алгоритмах важная роль отводится *рекурсии*, то есть вызову методом самого себя. Рекурсия рассматривается в главе 6, «Рекурсия». (Термин *метод* используется в Java; в других языках он заменяется терминами «функция», «процедура» или «подпрограмма».)

Определения

В этом разделе представлены некоторые термины, которые неоднократно встречаются в книге.

База данных

Термином *база данных* обозначается совокупность всех данных, используемых в некоторой ситуации. Предполагается, что все элементы базы данных имеют одинаковый формат. Например, в адресной книге, составленной из карточек, все карточки совместно образуют базу данных. Иногда в этом же значении используется термин *файл*.

Запись

Записями называются логические блоки, на которые делится база данных. Запись определяет формат хранения информации. В нашей аналогии с картотекой каждая отдельная карточка представляет запись. Запись включает всю информацию о некоторой сущности в ситуации, в которой существует много похожих сущностей. Запись может описывать человека в базе данных отдела кадров, деталь машины в базе данных складского учета, рецепт в файле поваренной книги и т. д.

Поле

Запись обычно делится на несколько полей. Поле содержит данные определенного типа. В адресной картотеке имя человека, его адрес или номер телефона является отдельным полем.

Более сложные программы баз данных используют записи с большим количеством полей. В записи, показанной на рис. 1.1, каждая строка представляет отдельное поле.

В Java (и других объектно-ориентированных языках) записи обычно представляются объектами соответствующих классов. Отдельные переменные в объектах представляют поля данных. Поля в объектах классов в языке Java называются *полями* (тогда как в других языках — таких, как C++ — используется термин «переменные экземпляров»).

Код работника:
Номер социального страхования:
Фамилия:
Имя:
Адрес:
Город:
Область:
Почтовый индекс:
Телефон:
Дата рождения:
Дата первого поступления на работу:
Зарплата:

Рис. 1.1. Запись с несколькими полями

Ключ

Для поиска записей в базе данных одно из полей записи назначается *ключом*. Запись ищется по заданному значению ключа. Например, в программе адресной книги поиск записи может производиться по значению ключевого поля имени «Браун». Если в результате поиска будет найдена запись с заданным ключом, вы можете обращаться ко всем ее полям, не только к ключевому полю. Можно сказать, что ключ «отпирает» всю запись. Для одного файла могут использоваться разные ключи — например, поиск может осуществляться как по номеру телефона, так и по полю адреса. Любое из полей на рис. 1.1 может использоваться в качестве ключа.

Объектно-ориентированное программирование

Этот раздел написан для читателей, которые еще не имели дела с объектно-ориентированным программированием. Сразу предупредим, что на нескольких страницах невозможно объяснить суть всех новаторских идей, относящихся к ООП. Мы всего лишь хотим, чтобы вы поняли примеры программ, приводимые позднее в тексте.

Если даже после чтения этого раздела и просмотра примеров вся эта затея с ООП останется такой же непонятной, как квантовая физика, вам потребуется более серьезный учебник.

Недостатки процедурных языков

Методология ООП была изобретена из-за того, что процедурные языки (C, Pascal, ранние версии BASIC), как выяснилось, плохо подходят для больших и сложных проектов. Почему, спросите вы?

Проблемы можно условно разделить на два типа: отсутствие соответствия между программой и реальным миром и внутренняя организация самой программы.

Плохое моделирование реального мира

Процедурные языки не позволяли качественно отразить концептуальную модель реальных ситуаций. Методы выполняют операции, данные хранят информацию, но большинство объектов реального мира делает и то и другое. Скажем, термостат на нагревателе выполняет операции (включение и выключение нагревателя), а также хранит информацию (текущая и предпочтительная температура).

Если написать программу управления термостатом на процедурном языке, скорее всего, в ней появятся два метода `furnace_on()` и `furnace_off()` и две глобальные переменные `currentTemp` (значение которой определяется показаниями термометра) и `desiredTemp` (значение задается пользователем). Однако эти методы и переменные

не формируют никакой программной единицы; в программе нет ничего, что можно было бы назвать общим термином thermostat. Такая концепция существует только в уме программиста.

В больших программах, содержащих тысячи подобных сущностей, процедурный подход повышал вероятность ошибок, устраивал хаос, а иногда и вовсе делал реализацию невозможной. Необходим был механизм определения более четких соответствий между объектами программы и объектами реального мира.

Примитивность организации

Более хитрая, хотя и взаимосвязанная проблема относилась к внутренней организации программ. В процедурных программах код делился на методы. Один из недостатков такого способа структурирования заключался в том, что методы выходили на первый план в ущерб данным. В том, что касалось данных, свобода выбора программиста была весьма ограничена. Слегка упрощая, можно сказать, что данные могли быть локальными для конкретного метода или же глобальными, то есть доступными для всех методов. Нельзя было (по крайней мере достаточно гибко и универсально) указать, что одни методы могут обращаться к переменной, а другим методам это запрещено.

Такое отсутствие гибкости создавало проблемы, когда разные методы должны были работать с одними данными. Чтобы сделать переменную доступной для нескольких методов, ее приходилось объявлять глобальной, однако глобальные данные становились доступными для *всех* методов программы. Это часто приводило к ошибкам программирования. Необходим был механизм точной настройки доступности, чтобы данные были доступны для тех методов, которым это действительно нужно, и оставались скрытыми от всех остальных методов.

Объекты в двух словах

Концепция объектов возникла в программном сообществе как средство решения проблем процедурных языков.

Объекты

В основу ООП был заложен революционный принцип: объект содержит *как* методы, *так и* переменные. Скажем, объект термостата мог содержать не только методы furnace_on() и furnace_off(), но и переменные currentTemp и desiredTemp. В языке Java такие переменные объектов называются *полями*.

Новая сущность — объект — решает сразу несколько проблем. Программные объекты не только лучше соответствуют объектам реального мира, но и решают проблемы глобальности данных в процедурной модели. Методы furnace_on() и furnace_off() могут обращаться к переменным currentTemp и desiredTemp. Однако эти переменные остаются скрытыми от методов, не входящих в класс термостата, что снижает вероятность их случайного изменения посторонним методом.

Классы

Сама концепция объекта достаточно революционна, но это еще не все. Довольно быстро пришло понимание того, что у программиста может возникнуть потребность в создании нескольких однотипных объектов. Допустим, вы пишете программу управления нагревателями для целого здания, и эта программа требует создания нескольких десятков объектов термостатов. Было бы крайне неудобно возиться с определением каждого объекта по отдельности. Так родилась идея классов.

Класс представляет собой спецификацию («чертеж») для создания одного или нескольких объектов. Например, определение класса термостата на языке Java выглядит примерно так:

```
class thermostat
   {
      private float currentTemp();
      private float desiredTemp();

      public void furnace_on()
      {
         // Тело метода
      }

      public void furnace_off()
      {
         // Тело метода
      }
   }  // Конец класса thermostat
```

Ключевое слово Java `class` объявляет начало спецификации класса; за ним следует имя, которое вы хотите присвоить классу (`thermostat` в нашем примере). В фигурные скобки заключены определения полей и методов, составляющих класс. Мы опустили тела методов; обычно каждый метод содержит много строк программного кода.

Программист C скажет, что это определение напоминает определение структур, а программист C++ — что оно очень похоже на определение класса C++, разве что в конце нет символа «;» (хотя так ли уж нужны эти символы в C++?).

Создание объектов

Определение класса не создает никаких объектов этого класса (по аналогии с тем, как определение структуры в коде C не создает никаких переменных). Чтобы создать объект в языке Java, необходимо использовать ключевое слово `new`. В момент создания объекта ссылка на него должна быть сохранена в переменной подходящего типа (то есть класса).

Что такое ссылка? Мы подробно рассмотрим ссылки позднее, а пока считайте, что это своего рода имя объекта. (На самом деле это адрес объекта, но вам пока это знать не обязательно.)

В следующем фрагменте мы определяем две ссылки на тип thermostat, создаем два объекта thermostat и сохраняем ссылки на них в этих переменных:

```
thermostat therm1, therm2;  // Определение двух ссылок
therm1 = new thermostat();  // Создание двух объектов
therm2 = new thermostat();  // Сохранение ссылок на них
```

Объекты также часто называются *экземплярами* классов, а процесс создания объекта — созданием экземпляра.

Вызов методов объекта

После того как вы определите класс и создадите объекты этого класса, какая-то другая часть программы будет взаимодействовать с этими объектами. Как это происходит?

Как правило, другие части программы взаимодействуют с методами объекта, а не с его данными (полями). Например, чтобы приказать объекту therm2 включить нагреватель, используйте синтаксис вида

```
therm2.furnace_on();
```

Оператор «точка» (.) связывает объект с одним из его методов (или полей данных).

На данный момент мы рассмотрели (более чем поверхностно) некоторые важнейшие особенности ООП. Подведем итог:

- Объекты состоят из методов и полей (данных).
- Класс представляет собой спецификацию для создания любого количества объектов.
- Объект создается ключевым словом new с указанием имени класса.
- Вызов метода для конкретного объекта осуществляется оператором «точка».

Концепции ООП глубоки и содержательны, а их применение имеет далеко идущие последствия. Их почти невозможно усвоить с первого раза; не огорчайтесь, если вы чего-то не поняли. Со временем вы увидите примеры классов и того, как они работают, и туман начнет рассеиваться.

Пример объектно-ориентированной программы

Ниже приведена объектно-ориентированная программа, которая реально работает и выдает результаты. Класс BankAccount моделирует текущий счет в банке. Программа создает счет, выводит баланс, осуществляет внесение и снятие средств, после чего выводит новый баланс. Код программы bank.java приведен в листинге 1.1.

Листинг 1.1. Программа bank.java

```
// bank.java
// Демонстрация базового синтаксиса ООП
// Запуск программы: C>java BankApp
```

продолжение ⇨

Листинг 1.1 (*продолжение*)

```
////////////////////////////////////////////////////////////////
class BankAccount
   {
   private double balance;                   // Баланс счета

   public BankAccount(double openingBalance) // Конструктор
      {
      balance = openingBalance;
      }

   public void deposit(double amount)        // Внесение средств
      {
      balance = balance + amount;
      }

   public void withdraw(double amount)       // Снятие средств
      {
      balance = balance - amount;
      }

   public void display()                     // Вывод баланса
      {
      System.out.println("balance=" + balance);
      }

   // Конец класса BankAccount
////////////////////////////////////////////////////////////////
class BankApp
   {
   public static void main(String[] args)

   {
      BankAccount ba1 = new BankAccount(100.00); // Создание счета

      System.out.print("Before transactions, ");
      ba1.display();                             // Вывод баланса

      ba1.deposit(74.35);                        // Внесение средств
      ba1.withdraw(20.00);                       // Снятие средств

      System.out.print("After transactions, ");
      ba1.display();                             // Вывод баланса
      }  // Конец main()
}  // Конец класса BankApp
```

Результат работы программы выглядит так:

```
Before transactions, balance=100
After transactions, balance=154.35
```

Программа bank.java состоит из двух классов. Первый — класс BankAccount — содержит поля и методы банковского счета (он будет подробно рассмотрен ниже). Второй класс — BankApp — играет особую роль.

Класс BankApp

Чтобы выполнить программу из листинга 1.1 в командной строке MS-DOS, введите команду java BankApp в приглашении C:

```
C:\>java BankApp
```

Эта команда приказывает интерпретатору java найти в классе BankApp метод с именем main(). Каждое приложение Java должно иметь метод main(); выполнение программы начинается с выполнения main(), как видно из листинга 1.1 (на аргумент String[] args метода main() пока не обращайте внимания).

Метод main() создает объект класса BankAccount и инициализирует его значением 100.00; обе операции выполняются следующей командой:

```
BankAccount ba1 = new BankAccount(100.00); // Создание счета
```

Метод System.out.print() выводит строку, передаваемую в аргументе (Before transactions:), а объект счета выводит текущий баланс следующей командой:

```
ba1.display();
```

Затем программа заносит и снимает средства со счета:

```
ba1.deposit(74.35);
ba1.withdraw(20.00);
```

Далее программа выводит новый баланс и завершает работу.

Класс BankAccount

Единственное поле данных класса BankAccount содержит текущую сумму на счету; этому полю присвоено имя balance. Класс содержит три метода. Метод deposit() вносит средства на счет, метод withdrawal() снимает средства со счета, а метод display() выводит баланс.

Конструкторы

Класс BankAccount также содержит *конструктор* — специальный метод, который вызывается автоматически при создании нового объекта. Имя конструктора всегда совпадает с именем класса, поэтому наш конструктор называется BankAccount(). Он получает один аргумент, который задает исходный баланс при создании счета.

Конструктор обеспечивает удобный механизм инициализации нового объекта. Без конструктора в программу пришлось бы включать дополнительный вызов deposit() для занесения на счет исходной суммы.

Public и private

Обратите внимание на ключевые слова public и private в классе BankAccount. Это *модификаторы доступа*, определяющие, какие методы могут обращаться к данному

методу или полю. Поле balance помечено ключевым словом private, то есть объявлено *приватным*. Приватное поле или метод доступны только для методов, принадлежащих тому же классу. Таким образом, команды main() не могут обращаться к полю balance, потому что метод main() не принадлежит BankAccount. Другие методы BankAccount помечены модификатором public (они называются *открытыми*), то есть доступны для методов других классов. Именно это обстоятельство позволяет методу main() вызывать deposit(), withdrawal() и display().

Поля данных класса обычно объявляются приватными, а методы — открытыми. Тем самым обеспечивается защита данных — их случайное изменение методами других классов. Любой внешний клиент, которому потребуется обратиться к данным классам, должен вызвать для этого метод того же класса.

Наследование и полиморфизм

В этом разделе кратко рассматриваются две ключевые особенности объектно-ориентированного программирования: наследование и полиморфизм.

Наследованием называется процесс создания одного (*расширенного* или *производного*) класса на основе другого класса, называемого *базовым* классом. Производный класс обладает всеми возможностями базового класса, а также некоторыми дополнительными возможностями. Например класс secretary (секретарь) может быть производным от более общего класса employee (работник) и в него может быть включено поле typingSpeed (скорость печати), которого не было в классе employee.

В языке Java наследование также называется *субклассированием*. Базовый класс может называться *суперклассом*, а производный класс — *субклассом*.

Наследование позволяет легко расширять функциональность существующего класса и является полезным инструментом при проектировании программ с множеством взаимосвязанных классов. Таким образом, наследование упрощает повторное использование классов для несколько иных целей — одно из важнейших преимуществ ООП.

Термином *полиморфизм* обозначается возможность одинаковой интерпретации объектов разных классов. Чтобы полиморфизм работал, эти разные классы должны быть производными от одного базового класса. На практике полиморфизм обычно сводится к вызову метода, при котором в зависимости от фактического класса конкретного объекта выполняются разные методы.

Например, вызов метода display() для объекта secretary приведет к вызову метода display класса secretary, тогда как вызов того же метода для объекта manager приведет к вызову другого метода display, определенного в классе manager. Полиморфизм упрощает и проясняет архитектуру и реализацию программы.

Для программистов, незнакомых с этими понятиями, наследование и полиморфизм создают немало дополнительных трудностей. Чтобы основное внимание в наших примерах было направлено на структуры данных и алгоритмы, мы старались не использовать их в наших программах. Безусловно, наследование и полиморфизм являются важными и очень полезными аспектами ООП, но для объяснения структур данных и алгоритмов они не обязательны.

Программотехника

В последние годы стало модно начинать книги, посвященные структурам данных и алгоритмам, с главы о программотехнике. Мы не стали следовать этой традиции, и все же необходимо хотя бы в общих чертах упомянуть программотехнику и ее отношение к темам, рассматриваемым в книге.

Программотехникой (software engineering) называется изучение способов создания больших и сложных программ с участием многих программистов. Основное внимание в программотехническом анализе направлено на общую архитектуру программ и на формирование этой архитектуры по потребностям конечных пользователей. Программотехника изучает жизненный цикл программных проектов, состоящий из фаз спецификации, проектирования, верификации, кодирования, тестирования, запуска и сопровождения.

Мы не рискнем утверждать, что смешение программотехники со структурами данных и алгоритмами поможет лучше понять любую из этих тем. Программотехника — довольно абстрактная дисциплина, которую трудно понять до того, как вы примете участие в крупном проекте. Использование структур данных и алгоритмов, с другой стороны, относится скорее к техническим дисциплинам, сопряженным с нюансами программирования и хранения данных.

Соответственно наше внимание будет сосредоточено на основных концепциях структур данных и алгоритмов. Как они работают? Какая структура или алгоритм лучше подходит для конкретной ситуации? Как они выглядят в коде Java? Еще раз подчеркнем, что наша задача — сделать материал как можно более понятным. За дополнительной информацией обращайтесь к книгам по программотехнике, перечисленным в приложении Б.

Java для программистов C++

Программистам C++, еще не знакомым с языком Java, стоит прочитать этот раздел. В нем описаны некоторые существенные отличия Java от C++.

Этот раздел не является учебником по языку Java. В нем даже не рассматриваются все отличия двух языков. Внимание уделяется лишь нескольким особенностям Java, из-за которых программисту C++ может быть трудно разобраться в примерах программ.

В Java нет указателей

Главное отличие C++ от Java заключается в том, что Java не использует указатели. Для программиста C++ это обстоятельство на первый взгляд выглядит довольно странно. Разве можно программировать без указателей?

В книге мы будем использовать программирование без указателей для построения сложных структур данных. Вы увидите, что код без указателей не только возможен — он еще и проще кода с указателями C++.

Выражаясь точнее, Java отказывается только от явных указателей. Указатели в форме адресов памяти остаются, хотя они и скрыты от программиста. Иногда говорят, что в Java нет ничего, кроме указателей. Нельзя сказать, что это утверждение полностью истинно, но оно близко к истине. А теперь подробнее...

Ссылки

В Java примитивные типы данных (такие как `int`, `float` или `double`) интерпретируются не так, как объекты. Рассмотрим две команды:

```
int intVar;        // Переменная int с именем intVar
BankAccount bc1;   // Ссылка на объект BankAccount
```

В первой команде область памяти, связанная с именем `intVar`, содержит числовое значение — например, 127 (если оно было в нее занесено). С другой стороны, область памяти `bc1` не содержит данные объекта `BankAccount`. В ней хранится адрес объекта `BankAccount`, находящегося где-то в другом месте памяти. Имя `bc1` ассоциировано со ссылкой на объект, а не с самим объектом.

Вообще говоря, область памяти `bc1` будет содержать ссылку лишь после того, как ей будет присвоен объект в ходе выполнения программы. До присваивания объекта в ней хранится ссылка на специальный объект с именем `null`. Аналогичным образом область памяти `intVar` не содержит числовое значение до того, как оно будет ей явно присвоено. Если попытаться использовать переменную, которой не было присвоено значение, компилятор выдаст сообщение об ошибке.

В C++ команда

```
BankAccount bc1;
```

выполняет фактическое создание объекта; при ее выполнении резервируется память, достаточная для хранения всех данных объекта. В языке Java такая команда всего лишь резервирует память, достаточную для хранения адреса объекта в памяти. Ссылку можно рассматривать как указатель с синтаксисом обычной переменной. (В C++ есть ссылочные переменные, но они должны быть явно объявлены при помощи символа `&`.)

Присваивание

Оператор присваивания (=) работает с объектами Java несколько иначе, чем с объектами C++. В C++ команда

```
bc2 = bc1;
```

копирует все данные из объекта `bc1` в другой объект `bc2`. После ее выполнения появляются два объекта с одинаковыми данными. Напротив, в Java эта команда присваивания копирует адрес памяти, на который ссылается `bc1`, в `bc2`. В результате оба имени, `bc1` и `bc2`, относятся к одному и тому же объекту, то есть являются ссылками на него.

Недостаточно хорошее понимание того, что делает оператор присваивания, может создать проблемы. После выполнения приведенной выше команды присваивания каждая из команд

```
bc1.withdraw(21.00);
```

и

```
bc2.withdraw(21.00);
```

снимет $21 с одного объекта банковского счета.

Допустим, вам потребовалось скопировать данные из одного объекта в другой. В таком случае придется позаботиться о том, чтобы в программе были созданы два разных объекта, и скопировать каждое поле по отдельности. Оператора присваивания для этого недостаточно.

Оператор new

Любой объект в Java должен быть создан оператором new. Однако в Java оператор new возвращает ссылку, а не указатель, как в C++ (то есть для использования new указатели тоже не нужны). Один из способов создания объекта:

```
BankAccount ba1;
ba1 = new BankAccount();
```

Устранение указателей повышает безопасность системы. Если программист не может узнать фактический адрес ba1 в памяти, он не сможет и случайно повредить данные объекта. С другой стороны, знать адрес объекта ему и не нужно, если только он не замышляет что-то нехорошее.

Как освободить память, которая была выделена оператором new, а теперь стала ненужной? В C++ используется оператор delete. В Java вообще не нужно беспокоиться об освобождении памяти. Java периодически просматривает все блоки памяти, выделенные оператором new, и проверяет, остались ли на них действительные ссылки. Если таких ссылок нет, блок возвращается в хранилище свободной памяти. Этот процесс называется *уборкой мусора*.

В C++ почти каждый программист время от времени забывал освобождать выделенную память, что приводило к «утечке памяти», неэффективному расходованию системных ресурсов, и как следствие — плохому быстродействию и даже сбоям системы. В Java утечка памяти невозможна (по крайней мере крайне маловероятна).

Аргументы

В C++ для передачи объектов функциям часто используются указатели; таким образом предотвращаются затраты ресурсов на копирование большого объекта. В Java объекты всегда передаются по ссылке, что также предотвращает копирование объекта:

```
void method1()
   {
   BankAccount ba1 = new BankAccount(350.00);
   method2(ba1);
   }

void method2(BankAccount acct)
   {
   }
```

В этом коде ссылки `bal` и `acct` относятся к одному объекту. В C++ `acct` будет отдельным объектом, созданным посредством копирования `bal`.

С другой стороны, примитивные типы данных всегда передаются по значению, то есть в методе создается новая переменная, в которую копируется значение аргумента.

Равенство и идентичность

Для примитивных типов языка Java оператор проверки равенства (==) определяет, содержат ли две переменные одинаковые значения:

```
int intVar1 = 27;
int intVar2 = intVar1;
if(intVar1 == intVar2)
System.out.println("They're equal");
```

В этом синтаксис Java не отличается от C и C+++, но поскольку в Java операторы сравнения используют ссылки, с объектами они работают иначе. Для объектов оператор проверки равенства проверяет идентичность двух ссылок, то есть относятся ли эти ссылки к одному объекту:

```
carPart cp1 = new carPart("fender");
carPart cp2 = cp1;
if(cp1 == cp2)
System.out.println("They're Identical");
```

В C++ этот оператор проверяет, содержат ли эти два объекта одинаковые данные. Для проверки этого условия в Java используется метод `equals()` класса `Object`:

```
carPart cp1 = new carPart("fender");
carPart cp2 = cp1;
if( cp1.equals(cp2) )
System.out.println("They're equal");
```

Этот прием работает, потому что все классы Java являются неявно производными от класса `Object`.

Перегрузка операторов

Здесь все просто: в Java перегрузка операторов не поддерживается. В C++ операторы +, *, = и многие другие можно переопределять, чтобы их смысл зависел от класса объектов, к которым они применяются. В Java такое переопределение невозможно. Замените операторы именованными методами, например `add()` вместо +.

Примитивные типы

Примитивные, или встроенные типы Java перечислены в табл. 1.2.

В отличие от языков C и C++, использующих целочисленное представление `true/false`, в Java существует специальный тип `boolean`.

Тип `char` является беззнаковым и состоит из двух байтов, обеспечивающих представление кодировки Юникод и поддержку национальных символов.

Таблица 1.2. Примитивные типы данных

Имя	Размер в битах	Диапазон значений
boolean	1	true или false
byte	8	от –128 до 127
char	16	от '\u0000' до '\uFFFF'
short	16	от –32,768 до +32,767
int	32	от –2 147 483 648 до +2 147 483 647
long	64	от –9 223 372 036 854 775 808 до +9 223 372 036 854 775 807
float	32	Приблизительно от 10^{-38} до 10^{+38}; 7 значащих цифр
double	64	Приблизительно от 10^{-308} до 10^{+308}; 15 значащих цифр

В C и C++ тип `int` имеет разные размеры в зависимости от конкретной компьютерной платформы; в Java тип `int` всегда занимает 32 бита.

Литералы типа `float` используют суффикс `F` (например, `3.14159F`); литералы типа `double` записываются без суффикса. Литералы типа `long` используют суффикс `L` (например, `45L`); литералы других целочисленных типов записываются без суффикса.

Язык Java обладает более сильной типизацией по сравнению с C и C++; многие преобразования, выполняемые в этих языках автоматически, должны явно записываться в Java.

Все типы, отсутствующие в табл. 1.2 (например, `String`), являются классами.

Ввод/вывод

По мере развития Java в подсистеме ввода/вывода произошли значительные изменения. В приложениях консольного режима, который будет использоваться в примерах книги, существуют некрасивые, но эффективные конструкции ввода/вывода, заметно отличающиеся от синтаксиса `cout/cin` языка C++ и `printf()/scanf()` языка C.

Старые версии Java SDK (Software Development Kit) требовали, чтобы для выполнения всех операций ввода/вывода в начало исходного файла включалась строка

```
import java.io.*;
```

Теперь эта строка необходима только для ввода.

Вывод

Для вывода всех примитивных типов (числовых и символьных), а также объектов `String` используются следующие вызовы методов:

```
System.out.print(var);   // Выводит значение var без перевода строки
System.out.println(var); // Выводит значение var с переходом на новую строку
```

Метод `print()` оставляет курсор в той же строке, а метод `println()` переводит его в начало следующей строки.

В старых версиях SDK конструкция `System.out.print()` ничего не выводила на экран — за ней должен был следовать вызов `System.out.println()` или `System.out.flush()`, который выводил содержимое всего буфера. Теперь данные выводятся на экран немедленно.

Методам вывода можно передавать несколько аргументов, разделенных знаком «+». Допустим, при выполнении следующей команды переменная `ans` содержит значение 33:

```
System.out.println("The answer is " + ans);
```

Результат выглядит так:

```
The answer is 33
```

Ввод строки

Ввод значительно сложнее вывода. Как правило, все входные данные должны читаться как объект `String`. Если вводятся данные другого типа (скажем, символ или число), объект `String` необходимо преобразовать к нужному типу.

Как упоминалось ранее, в начале любой программы, в которой используется ввод, должна стоять директива

```
import java.io.*;
```

Без этой директивы компилятор не распознает такие имена, как `IOException` и `InputStreamReader`.

Ввод строковых данных реализован довольно затейливо. Следующий метод возвращает строку, введенную пользователем:

```
public static String getString() throws IOException
   {
   InputStreamReader isr = new InputStreamReader(System.in);
   BufferedReader br = new BufferedReader(isr);
   String s = br.readLine();
   return s;
   }
```

Метод возвращает объект `String`, который строится из символов, введенных с клавиатуры и завершенных нажатием клавиши Enter. Классы `InputStreamReader` и `BufferedReader` нас сейчас не интересуют.

Кроме импортирования `java.io.*`, необходимо включить во все методы с вводом данных условие `throws IOException`, как это сделано в приведенном коде. Более того, условие `throws IOException` должно присутствовать в любом методе, вызывающем методы ввода, в том числе и в `main()`.

Ввод символа

Предположим, пользователь программы должен ввести символ (то есть ввести что-то и нажать Enter). Пользователь может ввести один символ или (по ошибке) несколько символов. Таким образом, самый безопасный способ чтения символа заключается в чтении объекта `String` и отделении его первого символа методом `charAt()`:

```
public static char getChar() throws IOException
   {
   String s = getString();
   return s.charAt(0);
   }
```

Метод charAt() класса String возвращает символ, находящийся в заданной позиции объекта String; в данном случае возвращается первый символ (индекс 0). Такое решение исключает нахождение во входном буфере лишних символов, которые могли бы создать проблемы с последующим вводом.

Ввод целых чисел

Чтобы прочитать число, следует создать объект String, как показано ранее, и привести его к нужному типу, используя метод преобразования. Метод getInt() преобразует ввод к типу int и возвращает его:

```
public int getInt() throws IOException
   {
   String s = getString();
   return Integer.parseInt(s);
   }
```

Метод parseInt() класса Integer преобразует строку к типу int. Аналогичный метод parseLong() используется для преобразования к типу long.

В старых версиях SDK в начале любой программы, использующей parseInt(), должна была присутствовать директива

```
import java.lang.Integer;
```

Сейчас эта директива не обязательна.

Для простоты мы не проверяем ошибки в методах ввода в примерах кода. Пользователь должен ввести правильное значение, иначе произойдет исключение. В представленном коде исключение приведет к завершению программы. Любая серьезная программа должна проверять входную строку перед попыткой преобразования, а также перехватывать исключения и обрабатывать их соответствующим образом.

Ввод вещественных чисел

Типы float и double обрабатываются примерно так же, как int, но процесс преобразования немного усложняется. Пример чтения типа double:

```
public int getDouble() throws IOException
   {
   String s = getString();
   Double aDub = Double.valueOf(s);
   return aDub.doubleValue();
   }
```

Сначала String преобразуется в объект типа Double (D в верхнем регистре!) — класса-«обертки» для примитивного значения double. Затем метод doubleValue() класса Double преобразует объект к типу double. Для типа float существует аналогичный класс Float с аналогичными методами valueOf() и floatValue().

Структуры данных библиотеки Java

Пакет `java.util` содержит такие структуры данных, как `Vector` (расширяемый массив), `Stack`, `Dictionary` и `Hashtable`. В этой книге встроенные классы почти не используются; мы учим читателя основам, а не тонкостям конкретной реализации. Впрочем, в отдельных случаях некоторые библиотечные структуры все же оказываются полезными. Перед использованием объектов этих классов в программу следует включить директиву

```
import java.util.*;
```

Впрочем, библиотеки классов — как встроенные в Java, так и предлагаемые сторонними разработчиками — содержат гибкие, хорошо отлаженные классы. Знания, которые вы получите в книге, помогут выбрать нужную структуру данных и понять, как она работает. И тогда вы уже сможете решить, то ли написать собственные классы, то ли воспользоваться уже готовыми.

Итоги

- ✦ Структурой данных называется способ организации данных в памяти компьютера или в дисковом файле.
- ✦ Правильный выбор структуры данных заметно повышает эффективность программы.
- ✦ Примеры структур данных — массивы, стеки и связанные списки.
- ✦ Алгоритм представляет собой процедуру выполнения некоторой операции.
- ✦ В Java алгоритм обычно реализуется методом класса.
- ✦ Многие структуры данных и алгоритмы, описанные в этой книге, часто используются для построения баз данных.
- ✦ Некоторые структуры данных используются как инструменты программиста: они упрощают реализацию алгоритма.
- ✦ Другие структуры данных моделируют ситуации из реального мира, например телефонные линии, проложенные между городами.
- ✦ Базой данных называется совокупность данных, состоящих из множества однотипных записей.
- ✦ Запись часто представляет объект из реального мира, например работника или деталь машины.
- ✦ Записи делятся на поля. В каждом поле хранится одна характеристика объекта, описываемого записью.
- ✦ Ключом называется поле записи, используемое для выполнения некоторой операции с данными, например сортировки записей работников по полю `LastName`.
- ✦ В базах данных можно найти все записи, у которых ключевое поле содержит заданное значение. Такое значение называется ключом поиска.

Вопросы

Следующие вопросы помогут читателю проверить качество усвоения материала. Ответы на них приведены в приложении В.

1. Многие структуры данных поддерживают операции ________ отдельных записей, ________ и ________ .
2. Перестановка содержимого структуры данных в определенном порядке называется ________.
3. В базе данных поле представляет собой:

   a) конкретный элемент данных;

   b) конкретный объект;

   c) часть записи;

   d) часть алгоритма.
4. Поле, используемое для поиска конкретной записи, называется ________ .
5. В объектно-ориентированном программировании объект:

   a) эквивалентен классу;

   b) может содержать даты и методы;

   c) является программой;

   d) может содержать классы.
6. Класс:

   a) представляет собой «шаблон» для создания объектов;

   b) представляет конкретный объект реального мира;

   c) хранит конкретные значения в своих полях;

   d) задает тип метода.
7. В Java спецификация класса:

   a) создает объекты;

   b) содержит ключевое слово `new`;

   c) создает ссылки;

   d) ни одно из приведенных утверждений не верно.
8. Когда объекту требуется выполнить операцию, он использует ________ .
9. В Java для обращения к методам объекта необходимо использовать оператор ______.
10. В Java `boolean` и `byte` являются ________ .

    (В главе 1 нет ни упражнений, ни программных проектов.)

Глава 2

Массивы

Самая распространенная структура данных — массив — поддерживается в большинстве языков программирования. Вследствие своей популярности массив является хорошей отправной точкой для знакомства со структурами данных и изучения их роли в объектно-ориентированном программировании. В этой главе мы рассмотрим массивы языка Java, а также разработаем свою, доморощенную реализацию массива.

Также будет рассмотрена особая разновидность массивов — упорядоченный массив, в котором данные хранятся в порядке возрастания (или убывания) ключа. Такой способ хранения данных позволяет использовать механизм быстрого поиска элементов — двоичный поиск.

Глава начинается с создания приложения Java Workshop, демонстрирующего операции вставки, поиска и удаления в массиве. Затем будут представлены примеры выполнения тех же операций в коде Java.

Далее мы перейдем к упорядоченным массивам. Начнем, как и в предыдущем случае, с приложения Workshop и рассмотрим пример двоичного поиска. Глава завершается описанием «O-синтаксиса» — самой распространенной метрики эффективности алгоритмов.

Приложение Array Workshop

Допустим, вы работаете тренером детской бейсбольной команды и вам требуется хранить информацию об игроках, находящихся на поле. Вам нужна программа для ноутбука — программа с базой данных игроков, вышедших на тренировку. Для хранения списка достаточно простой структуры данных. Вероятно, с данными будут выполняться несколько основных операций:

- Вставка в структуру данных сведений об игроке, выходящем на поле.
- Проверка наличия сведений о конкретном игроке (поиск игрока в структуре по номеру).
- Удаление из структуры данных игрока, покинувшего поле.

Эти три операции — вставка, поиск, удаление — занимают центральное место в большинстве структур данных, которые будут рассматриваться в книге.

Описание конкретной структуры данных часто будет начинаться с демонстрации на примере приложения Workshop. Сначала читатель получает практическое

представление о том, как работает структура данных и ее алгоритмы, а затем мы переходим к подробному описанию и примерам кода. Приложение с именем Array Workshop покажет, как операции вставки, поиска и удаления выполняются с массивами.

Запустите приложение Array Workshop (см. приложение A, «Приложения Workshop и примеры программ») командой

```
C:\>appletviewer Array.html
```

На рис. 2.1 показан массив с 20 элементами, 10 из которых содержат данные. Каждый элемент представляет одного конкретного игрока. Представьте, что игроки носят футболки и на спине у каждого указан номер игрока. А чтобы картина лучше выглядела, игроки носят футболки разных цветов. В приложении видны как номера игроков, так и цвета футболок.

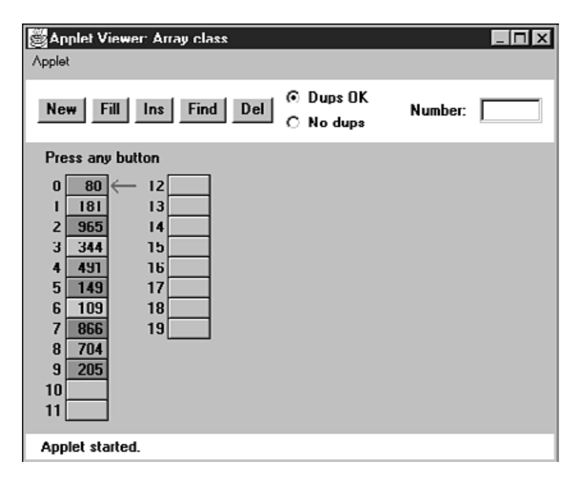


Рис. 2.1. Приложение Array Workshop

Приложение демонстрирует все три основные операции, о которых упоминалось ранее:

- Кнопка `Ins` вставляет новый элемент данных.
- Кнопка `Find` ищет элемент данных в массиве.
- Кнопка `Del` удаляет элемент данных.

Кнопка `New` создает новый массив заданного размера, а кнопка `Fill` заполняет его любым количеством элементов данных. Кнопка `Fill` создает набор элементов и присваивает им случайные номера в диапазоне от 0 до 999 и цвета. Приложение не позволяет создать массив, содержащий более 60 ячеек, и разумеется, количество заполняемых элементов данных не может превысить количество ячеек в массиве.

Кроме того, при создании нового массива необходимо решить, может ли массив содержать дубликаты; вскоре мы вернемся к этому вопросу. По умолчанию дубликаты запрещены, поэтому переключатель `No Dups` изначально установлен.

Вставка

Начнем с конфигурации по умолчанию: 20 ячеек и 10 элементов данных, дубликаты запрещены (`No Dups`). Когда новый игрок выходит на поле, его номер вставляется в массив. Чтобы вставить новый элемент, нажмите кнопку `Ins`. Вам будет предложено ввести значение элемента:

```
Enter key of item to insert
```

Введите число (скажем 678) в текстовом поле, расположенном в правом верхнем углу приложения. (Да, трехзначные номера на футболках встречаются редко.) Снова нажмите кнопку `Ins`; приложение подтвердит ваш выбор:

```
Will insert item with key 678
```

Последнее нажатие кнопки вставляет элемент данных с заданным значением и случайным цветом в первую пустую ячейку массива. Сообщение будет выглядеть примерно так:

```
Inserted item with key 678 at index 10
```

Каждое нажатие кнопки в приложении Workshop соответствует одному из шагов реализации алгоритма. Чем больше шагов, тем больше времени занимает выполнение алгоритма. В приложении Array вставка выполняется очень быстро и состоит всего из одного шага. Новый элемент всегда вставляется в первую свободную ячейку массива, причем алгоритму уже известен номер этой ячейки, так как он знает текущее количество элементов в массиве. Операции поиска и удаления выполняются не так быстро.

В режиме запрета дубликатов необходимо следить за тем, чтобы в массив не вставлялись элементы, у которых значение ключа совпадает со значением ключа существующего элемента. Приложение выводит сообщение об ошибке, но не отменяет вставку. Предполагается, что пользователь сам должен следить за правильностью данных.

Поиск

Чтобы выполнить поиск, щелкните на кнопке `Find`. Вам будет предложено ввести номер (ключ) искомого игрока. Введите номер элемента, находящегося где-то в середине массива, и многократно нажимайте кнопку `Find`. При каждом нажатии кнопки выполняется один шаг алгоритма. Вы увидите, как красная стрелка начинает последовательно перемещаться от ячейки 0, проверяя текущую ячейку при каждом нажатии. Индекс ячейки в сообщении изменяется в процессе перемещения:

```
Checking next cell, index = 2
```

Когда элемент с заданным ключом будет найден, появляется сообщение

```
Have found item with key 505
```

(или тем значением ключа, которое вы ввели). Если дубликаты запрещены, то поиск завершится при обнаружении элемента с заданным значением ключа.

Если выбранное значение ключа отсутствует в массиве, приложение проверяет все заполненные ячейки, а потом сообщает, что значение найти не удалось. Обратите внимание: алгоритм поиска в среднем должен просмотреть половину элементов данных в массиве (предполагается, что дубликаты запрещены). Элементы, расположенные вблизи от начала массива, будут найдены быстрее, а элементы в конце массива — позднее. Если массив содержит *N* элементов, то среднее количество шагов, необходимых для нахождения элемента, равно *N*/2. В худшем случае заданный элемент находится в последней ячейке и для его обнаружения потребуется *N* шагов.

Таким образом, время выполнения алгоритма пропорционально количеству шагов, а поиск в среднем выполняется значительно медленнее (*N*/2 шагов) вставки (один шаг).

Удаление

Чтобы удалить элемент, необходимо сначала найти его. После ввода номера удаляемого игрока повторные нажатия кнопки последовательно перемещают стрелку, пока не будет найден заданный элемент. Следующее нажатие кнопки удаляет элемент, и ячейка становится пустой. (Строго говоря, эта операция не обязательна, потому что в ячейку все равно будет записано другое значение, но удаление четко показывает, что сейчас происходит.)

Алгоритм удаления подразумевает, что массив не может содержать пустых элементов, то есть пустых ячеек, за которыми находятся заполненные ячейки (с более высокими значениями индексов). Наличие «дыр» усложняет все алгоритмы, потому что перед проверкой содержимого ячейки они должны сначала проверить, не пуста ли она. Кроме того, необходимость проверки незанятых ячеек снижает эффективность алгоритмов. По этим причинам расположение занятых ячеек должно быть смежным, то есть без «дыр».

А это значит, что после обнаружения заданного элемента и его удаления приложение должно сдвинуть все последующие элементы на одну позицию к началу массива, чтобы заполнить пустой элемент. Пример показан на рис. 2.2.

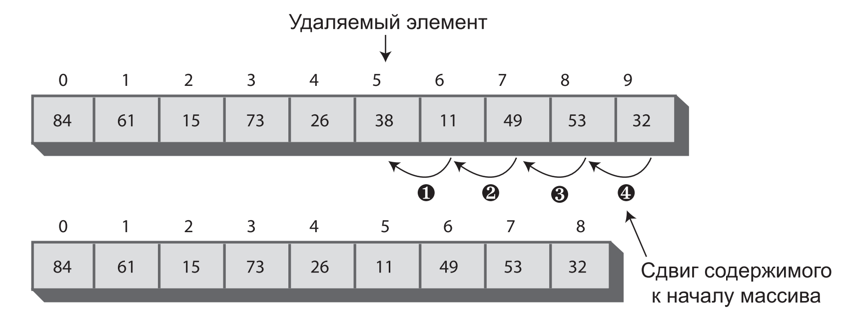


Рис. 2.2. Удаление элемента

Так, при удалении элемента в ячейке 5 (номер 38 на рис. 2.2) элемент в ячейке 6 сдвигается в ячейку 5, элемент в ячейке 7 сдвигается в ячейку 6 и так далее до последней занятой ячейки. В процессе удаления приложение сдвигает содержимое последующих ячеек при нажатии кнопки Del.

Удаление (при условии запрета дубликатов) требует поиска в среднем по $N/2$ элементам и последующего перемещения остальных элементов (в среднем $N/2$) для заполнения пустой ячейки. В сумме получается N шагов.

Проблема дубликатов

При проектировании структуры данных необходимо решить, допустимо ли существование элементов с одинаковыми ключами. Скажем, если вы работаете с базой данных работников, а ключом является табельный номер, дубликаты не имеют смысла — два работника не должны иметь одинаковые табельные номера. С другой стороны, если ключевым является поле фамилии, его значения вполне могут совпадать у нескольких работников, так что дубликаты запрещать нельзя.

Конечно, номера игроков на бейсбольном поле не должны повторяться, однако приложение Array Workshop позволяет опробовать оба режима. При создании нового массива кнопкой New приложение предлагает задать размер и указать, разрешены ли дубликаты. Решение принимается при помощи переключателей Dups OK и No Dups.

Если в вашей программе дубликаты запрещены, вероятно, вам стоит предусмотреть защиту от ошибок ввода данных при вставке — проверьте все элементы массива и убедитесь в том, что их ключевые значения не совпадают с ключевым значением вставляемого элемента. Однако такая проверка весьма неэффективна, к тому же она повышает количество шагов, необходимых для выполнения вставки, с 1 до N. По этой причине в нашем приложении такая проверка не выполняется.

Поиск с дубликатами

Как было замечено ранее, дубликаты усложняют алгоритм поиска. Даже при обнаружении совпадения необходимо продолжить поиск дополнительных совпадений до последней занятой ячейки. Впрочем, это не единственный вариант — поиск также можно прервать после первого совпадения. Выбор зависит от того, какой вопрос вас интересует: «найдите мне записи всех людей с голубыми глазами» или «найдите мне запись любого человека с голубыми глазами».

При установленном переключателе Dups OK приложение выбирает первый вариант и ищет все записи с подходящим ключом сортировки. Поиск всегда выполняется за N шагов, потому что алгоритм должен перебрать все содержимое массива до самой последней ячейки.

Вставка с дубликатами

Вставка выполняется независимо от того, разрешены дубликаты или нет: новый элемент вставляется за один шаг. Но помните: если дубликаты запрещены, а пользователь может случайно ввести один и тот же ключ дважды, вероятно, вам придется проверять все существующие элементы перед вставкой.

Удаление с дубликатами

Удаление с разрешенными дубликатами может усложниться — все зависит от того, что именно подразумевается под «удалением». При удалении только первого элемента с заданным значением в среднем придется выполнить только $N/2$ сравнений и $N/2$ перемещений. В этом случае сложность удаления не отличается от сложности удаления без дубликатов.

Но если под удалением подразумевается удаление *всех* элементов с заданным ключом, то, возможно, удалять придется сразу несколько элементов. Такая операция потребует проверки N ячеек и (возможно) перемещения более чем $N/2$ ячеек. Средняя оценка зависит от распределения дубликатов в массиве.

Приложение выбирает второй вариант: удаляются все элементы с одинаковым значением ключа. Задача усложняется тем, что при каждом удалении последующие элементы должны сдвигаться дальше. Например, если из массива удаляются три ячейки, то элементы за последней ячейкой придется сдвинуть на три позиции. Чтобы понять, как работает эта операция, выберите в приложении режим `Dups OK`, вставьте три-четыре элемента с одинаковым ключом и удалите их.

В таблице 2.1 приведено среднее количество сравнений и перемещений для всех трех операций — сначала без дубликатов, а затем с дубликатами. N — количество элементов в массиве. Вставка нового элемента считается за одно перемещение.

Таблица 2.1. Среднее количество сравнений и перемещений

Операция	Без дубликатов	С дубликатами
Поиск	$N/2$ сравнений	N сравнений
Вставка	Без сравнений, одно перемещение	Без сравнений, одно перемещение
Удаление	$N/2$ сравнений, $N/2$ перемещений	N сравнений, более $N/2$ перемещений

Поэкспериментируйте с этими операциями при помощи приложения Array Workshop.

Различия между N и $N/2$ обычно считаются незначительными, если только вы не занимаетесь оптимизацией программы. Как будет показано ближе к концу главы, существуют более принципиальные различия: выполняется ли операция за один шаг, за N шагов, за $\log(N)$ или N^2 шагов.

Не слишком быстро

Работая с приложением Array Workshop, обратите внимание на медленную, педантичную работу алгоритмов. Не считая вставки, алгоритмы требуют перебора некоторых или почти всех ячеек массива. Другие структуры данных предоставляют намного более быстрые (но и более сложные) алгоритмы. Один из таких алгоритмов — двоичный поиск в упорядоченном массиве — будет представлен позднее в этой главе, а другие примеры встречаются в других главах.

Поддержка массивов в Java

В предыдущем разделе наглядно продемонстрированы алгоритмы основных операций с массивами. Прежде чем переходить к программной реализации этих алгоритмов, мы сначала познакомимся с некоторыми основными положениями массивов Java.

Знатоки Java могут сразу переходить к следующему разделу, а программистам C и C++ стоит задержаться. Синтаксис массивов Java близок к синтаксису C и C++ (и имеет много общего с другими языками), однако у поддержки массивов Java есть свои специфические особенности.

Создание массива

Как упоминалось в главе 1, «Общие сведения», данные Java делятся на две категории: примитивные типы (`int`, `double` и т. д.) и объекты. Во многих языках программирования (даже в объектно-ориентированных, как C++) массивы считаются примитивными типами, но в Java они относятся к объектам. Соответственно для создания массива необходимо использовать оператор `new`:

```
int[] intArray;              // Определение ссылки на массив
intArray = new int[100];     // Создание массива
                             // и сохранение ссылки на него в intArray
```

Также можно воспользоваться эквивалентной объединенной конструкцией:

```
int[] intArray = new int[100];
```

Оператор `[]` сообщает компилятору, что имя принадлежит объекту массива, а не обычной переменной. В альтернативной форме этого оператора квадратные скобки ставятся после имени, а не после типа:

```
int intArray[] = new int[100];  // Альтернативный синтаксис
```

Однако размещение `[]` после `int` ясно показывает, что `[]` является частью типа, а не имени.

Поскольку массив является объектом, его имя (`intArray` в предшествующем коде) содержит *ссылку* на массив. Сам массив хранится где-то в памяти, а `intArray` содержит только адрес блока данных.

Массивы содержат поле `length`, которое может использоваться для определения размера (количества элементов) массива:

```
int arrayLength = intArray.length;   // Определение размера массива
```

Как и в большинстве языков программирования, размер уже созданного массива изменить невозможно.

Обращение к элементам массива

При обращении к элементу массива необходимо указать его индекс в квадратных скобках. Этот синтаксис также используется во многих других языках программирования:

```
temp = intArray[3];  // Получение содержимого четвертого элемента массива
intArray[7] = 66;    // Вставка значения 66 в восьмую ячейку
```

Следует помнить, что в Java, как и в C и C++, индекс первого элемента равен 0, так что индексы массива из 10 элементов лежат в диапазоне от 0 до 9.

Если индекс меньше 0 или больше размера массива без 1, генерируется исключение выхода за границы массива.

Инициализация

Если в программе явно не указано иное, массив целых чисел автоматически инициализируется 0 при создании. В отличие от C++ это относится даже к массивам, определяемым внутри методов (функций). Допустим, вы создаете массив объектов следующего вида:

```
autoData[] carArray = new autoData[4000];
```

Пока элементам массива не будут присвоены явные значения, они содержат специальный объект `null`. При попытке обращения к элементу массива, содержащему `null`, происходит ошибка присваивания `null`. Мораль: прежде чем обращаться к элементу, позаботьтесь о том, чтобы ему было присвоено значение.

Для инициализации массива примитивного типа значениями, отличными от 0, используется следующий синтаксис:

```
int[] intArray = { 0, 3, 6, 9, 12, 15, 18, 21, 24, 27 };
```

Эта команда совмещает объявление ссылки с созданием массива оператором `new` (хотя на первый взгляд это и не очевидно). Числа в фигурных скобках называются списком инициализации. Размер массива определяется по количеству значений в этом списке.

Пример массива

В этом разделе приведены примеры использования массивов в программах. Сначала мы рассмотрим традиционную процедурную версию, а затем перейдем к объектно-ориентированному аналогу. В листинге 2.1 представлена традиционная версия `array.java`.

Листинг 2.1. Программа array.java

```
// array.java
// Работа с массивами Java
// Запуск программы:  C>java arrayApp
////////////////////////////////////////////////////////////////
class ArrayApp
   {
   public static void main(String[] args)
      {
      long[] arr;                  // Ссылка на массив
```

продолжение ⇨

Листинг 2.1 (*продолжение*)

```
      arr = new long[100];          // Создание массива
      int nElems = 0;               // Количество элементов
      int j;                        // Счетчик цикла
      long searchKey;               // Ключи искомого элемента
//--------------------------------------------------------------
      arr[0] = 77;                  // Вставка 10 элементов
      arr[1] = 99;
      arr[2] = 44;
      arr[3] = 55;
      arr[4] = 22;
      arr[5] = 88;
      arr[6] = 11;
      arr[7] = 00;
      arr[8] = 66;
      arr[9] = 33;
      nElems = 10;                  // Массив содержит 10 элементов
//--------------------------------------------------------------
      for(j=0; j<nElems; j++)      // Вывод элементов
         System.out.print(arr[j] + " ");
      System.out.println("");
//--------------------------------------------------------------
      searchKey = 66;               // Поиск элемента с ключом 66
      for(j=0; j<nElems; j++)           // Для каждого элемента
         if(arr[j] == searchKey)        // Значение найдено?
            break;                      // Да - выход из цикла
      if(j == nElems)                   // Достигнут последний элемент?
         System.out.println("Can't find " + searchKey); // Да
      else
         System.out.println("Found " + searchKey);      // Нет
//--------------------------------------------------------------
   searchKey = 55;                      // Удаление элемента с ключом 55
   for(j=0; j<nElems; j++)              // Поиск удаляемого элемента
   if(arr[j] == searchKey)
      break;
   for(int k=j; k<nElems-1; k++)        // Сдвиг последующих элементов
      arr[k] = arr[k+1];
   nElems--;                            // Уменьшение размера
//--------------------------------------------------------------
   for(j=0; j<nElems; j++)              // Вывод элементов
      System.out.print( arr[j] + " ");
   System.out.println("");
   }
}  // Конец класса ArrayApp
```

Программа создает массив с именем arr, помещает в него 10 элементов данных (номера игроков), ищет элемент с ключом 66, выводит все элементы, удаляет игрока с ключом 55 и выводит оставшиеся 9 элементов. Результат работы программы выглядит так:

```
77 99 44 55 22 88 11 0 66 33
Found 66
77 99 44 22 88 11 0 66 33
```

В массиве сохраняются данные типа `long`. Этот тип выбран для того, чтобы более наглядно показать, что это данные; тип `int` используется для значений индексов. Примитивный тип выбран для упрощения кода. В общем случае данные, сохраняемые в структуре данных, состоят из нескольких полей, поэтому для их представления обычно используются объекты вместо примитивных типов. Пример будет приведен ближе к концу главы.

Вставка

Вставка нового элемента в массив выполняется просто — с использованием обычного синтаксиса массива:

```
arr[0] = 77;
```

Количество элементов, добавленных в массив, хранится в переменной `nElems`.

Поиск

Переменная `searchKey` содержит искомое значение. Чтобы найти нужный элемент, мы перебираем содержимое массива, сравнивая текущий элемент с `searchKey`. Если переменная цикла `j` достигает последней занятой ячейки, а совпадение так и не найдено, значит, значение отсутствует в массиве. В зависимости от результата поиска программа выводит соответствующее сообщение (`Found 66` или `Can't find 27`).

Удаление

Удаление начинается с поиска заданного элемента. Для простоты мы (оптимистично) предполагаем, что элемент в массиве присутствует. Обнаружив его, мы сдвигаем все элементы с большими значениями индекса на один элемент вниз, чтобы заполнить «дыру», оставшуюся от удаленного элемента, а значение `nElems` уменьшается на 1. В реальной программе также необходимо предусмотреть действия на тот случай, если удаляемое значение отсутствует в массиве.

Вывод

Операция вывода всех элементов весьма тривиальна: программа перебирает содержимое массива и последовательно выводит каждый текущий элемент `arr[j]`.

Структура программы

Структура программы `array.java` оставляет желать лучшего. Программа содержит единственный класс `ArrayApp`, а этот класс имеет только один метод `main()`. По сути, программа `array.java` написана в традиционном процедурном стиле. Давайте посмотрим, не сделает ли объектно-ориентированная реализация ее более понятной (и какие еще преимущества она принесет).

Переход на объектно-ориентированный стиль будет происходить в два этапа. На первом этапе мы отделим структуру данных (массив) от остального кода программы. Оставшаяся часть программы становится *пользователем* этой структуры. На втором этапе будет усовершенствовано взаимодействие между структурой хранения данных и ее пользователем.

Деление программы на классы

Программа `array.java` в листинге 2.1 фактически состоит из одного большого метода. Деление программы на классы обладает многочисленными преимуществами. Какие классы? Собственно структура данных и остальная часть программы, использующая эту структуру данных. Разделение программы на два класса проясняет ее функциональность, упрощает ее проектирование и понимание (а также модификацию и сопровождение кода в реальных проектах).

В программе `array.java` массив использовался в качестве структуры данных, но он рассматривался просто как элемент языка. В новой версии массив будет инкапсулирован в классе с именем `LowArray`. Класс будет предоставлять методы, при помощи которых объекты других классов (`LowArrayApp` в данном случае) смогут обращаться к массиву. Эти методы обеспечивают взаимодействие между `LowArray` и `LowArrayApp`. Первая версия класса `LowArray` не идеальна, но она наглядно доказывает необходимость в совершенствовании архитектуры. Код программы `lowArray.java` представлен в листинге 2.2.

Листинг 2.2. Программа lowArray.java

```
// lowArray.java
// Класс массива с низкоуровневым интерфейсом
// Запуск программы: C>java LowArrayApp
////////////////////////////////////////////////////////////////
class LowArray
   {
   private long[] a;                 // Ссылка на массив a
//--------------------------------------------------------------
   public LowArray(int size)         // Конструктор
      { a = new long[size]; }        // Создание массива
//--------------------------------------------------------------
   public void setElem(int index, long value)     // Запись элемента
      { a[index] = value; }
//--------------------------------------------------------------
   public long getElem(int index)                 // Чтение элемента
      { return a[index]; }
//--------------------------------------------------------------
   }  // Конец класса LowArray
////////////////////////////////////////////////////////////////
class LowArrayApp
   {
      public static void main(String[] args)
      {
```

```
      LowArray arr;                 // Ссылка
      arr = new LowArray(100);      // Создание объекта LowArray
      int nElems = 0;               // Количество элементов в массиве
      int j;                        // Переменная цикла

      arr.setElem(0, 77);           // Вставка 10 элементов
      arr.setElem(1, 99);
      arr.setElem(2, 44);
      arr.setElem(3, 55);
      arr.setElem(4, 22);
      arr.setElem(5, 88);
      arr.setElem(6, 11);
      arr.setElem(7, 00);
      arr.setElem(8, 66);
      arr.setElem(9, 33);
      nElems = 10;                 // Массив содержит 10 элементов

      for(j=0; j<nElems; j++)      // Вывод элементов
         System.out.print(arr.getElem(j) + " ");
      System.out.println("");

      int searchKey = 26;          // Поиск элемента
      for(j=0; j<nElems; j++)            // Для каждого элемента
         if(arr.getElem(j) == searchKey) // Значение найдено?
            break;
      if(j == nElems)                    // Нет
         System.out.println("Can't find " + searchKey);
      else                               // Да
         System.out.println("Found " + searchKey);

                         // Удаление элемента с ключом 55
      for(j=0; j<nElems; j++)            // Поиск удаляемого элемента
      if(arr.getElem(j) == 55)
         break;
      for(int k=j; k<nElems; k++)        // Сдвиг последующих элементов
         arr.setElem(k, arr.getElem(k+1) );
      nElems--;                          // Уменьшение размера

      for(j=0; j<nElems; j++)      // Вывод содержимого
         System.out.print( arr.getElem(j) + " ");
      System.out.println("");
      }
   }  // Конец класса LowArrayApp
////////////////////////////////////////////////////////////////
```

Результат выполнения программы `lowArray.java` напоминает результаты `array.java`, не считая того, что перед удалением элемента с ключевым значением 55 мы пытаемся найти несуществующее ключевое значение (26):

```
77 99 44 55 22 88 11 0 66 33
Can't find 26
77 99 44 22 88 11 0 66 33
```

Классы LowArray и LowArrayApp

В программе `lowArray.java` класс `LowArray` фактически представляет собой «обертку» для обычного массива Java. Массив скрыт от окружающего мира внутри класса; поле данных объявлено с ключевым словом `private`, так что оно доступно только для методов класса `LowArray`. Класс `LowArray` содержит три метода: `setElem()` и `getElem()` выполняют вставку и выборку элемента соответственно, а конструктор создает пустой массив заданного размера.

Другой класс, `LowArrayApp`, создает объект класса `LowArray` и использует его для хранения и обработки данных. Считайте, что `LowArray` — инструмент, а `LowArrayApp` — пользователь инструмента. Программа разделена на два класса с четко определенными ролями; это важный первый шаг на пути перехода к объектно-ориентированной архитектуре.

Класс, используемый для хранения объектов данных (как класс `LowArray` в программе `lowArray.java`), иногда называется контейнерным классом (или просто *контейнером*). Как правило, контейнер не только хранит данные, но и предоставляет методы для обращения к ним; также могут предоставляться методы для выполнения сортировки и других сложных операций.

Интерфейсы классов

Итак, программу можно разделить на классы. Как эти классы взаимодействуют друг с другом? Коммуникации между классами и разделение обязанностей между ними являются важными аспектами объектно-ориентированного программирования.

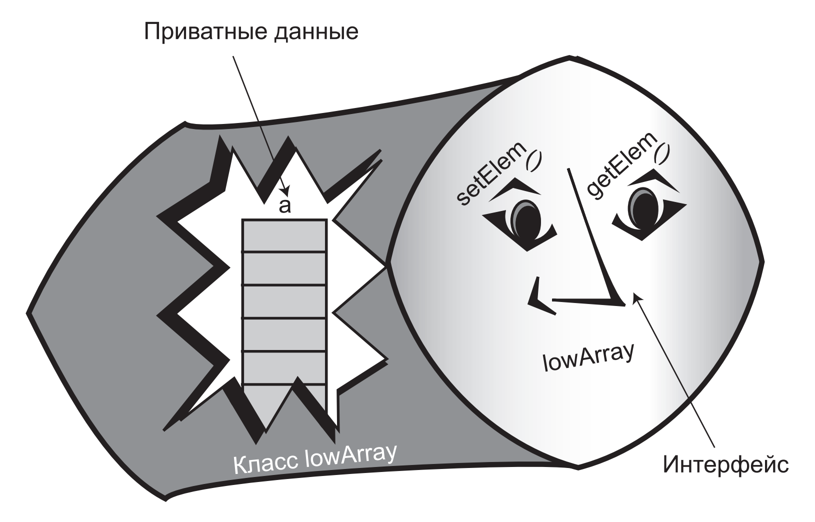


Рис. 2.3. Интерфейс LowArray

Они особенно важны для классов, имеющих нескольких пользователей. Как правило, желательно, чтобы классы могли использоваться снова и снова разными

пользователями (или одним и тем же пользователем) для разных целей. Например, кто-то может воспользоваться классом LowArray в другой программе для хранения номеров своих дорожных чеков. Класс справится с этой задачей так же хорошо, как с хранением номеров игроков.

Если класс будет использоваться разными программистами, его необходимо спроектировать так, чтобы с ним было удобно работать. Способ взаимодействия пользователя с классом называется *интерфейсом* класса. Поскольку поля класса обычно объявляются приватными, под интерфейсом подразумеваются методы класса — их функциональность и аргументы. Вызывая эти методы, пользователь класса взаимодействует с объектом этого класса. Одно из важных преимуществ объектно-ориентированного программирования состоит в том, что оно позволяет сделать интерфейс класса максимально удобным и эффективным. На рис. 2.3 показана графическая интерпретация интерфейса LowArray.

Не слишком удобно

Интерфейс класса LowArray в программе lowArray.java не отличается удобством. Методы setElem() и getElem() работают на слишком низком концептуальном уровне — по сути, они делают абсолютно то же, что делает оператор [] в обычных массивах Java. Пользователю класса, представленному методом main() класса LowArrayApp, в конечном итоге придется выполнять те же низкоуровневые операции, что и с традиционной версией массива в программе array.java. Единственное отличие заключается в том, что пользователь имеет дело с методами setElem() и getElem() вместо оператора []. В целом у такого подхода нет сколько-нибудь существенных преимуществ.

Также обратите внимание на отсутствие удобных средств вывода содержимого массива. Классу LowArrayApp приходится несколько неуклюже вызывать метод getElem() в цикле for. Дублирования кода можно было бы избежать, определив в LowArrayApp отдельный метод для вывода содержимого массива, но в какой степени этот метод относится к обязанностям класса LowArrayApp?

Таким образом, класс lowArray.java демонстрирует деление программы на классы, но мало что дает в практическом смысле. Давайте посмотрим, как правильно распределить обязанности между классами, чтобы лучше использовать преимущества ООП.

Кто чем занимается?

В программе lowArray.java метод main() класса LowArrayApp (пользователь структуры данных) должен отслеживать индексы массива. Для пользователей класса, которые нуждаются в произвольном доступе к элементам и не возражают против хранения индексов, такое решение имеет смысл. Например, как будет показано в следующей главе, оно может эффективно использоваться при сортировке массива.

Однако в типичной программе индексы массива не нужны пользователю.

Пример highArray.java

В следующей программе интерфейс класса хранения данных (здесь он называется `HighArray`) усовершенствован. Пользователю новой версии класса (`HighArrayApp`) уже не нужно думать об индексах. Методы `setElem()` и `getElem()` исчезли; на смену им пришли методы `insert()`, `find()` и `delete()`. Новые методы не требуют передачи индекса в аргументе, потому что класс берет обработку индексов на себя. Пользователь класса (`HighArrayApp`) может направить усилия на то, *что* нужно сделать (какие элементы необходимо вставить, удалить или получить), а не на то, *как* выполнить эти операции.

На рис. 2.4 изображен интерфейс `HighArray`, а в листинге 2.3 приводится код программы `highArray.java`.

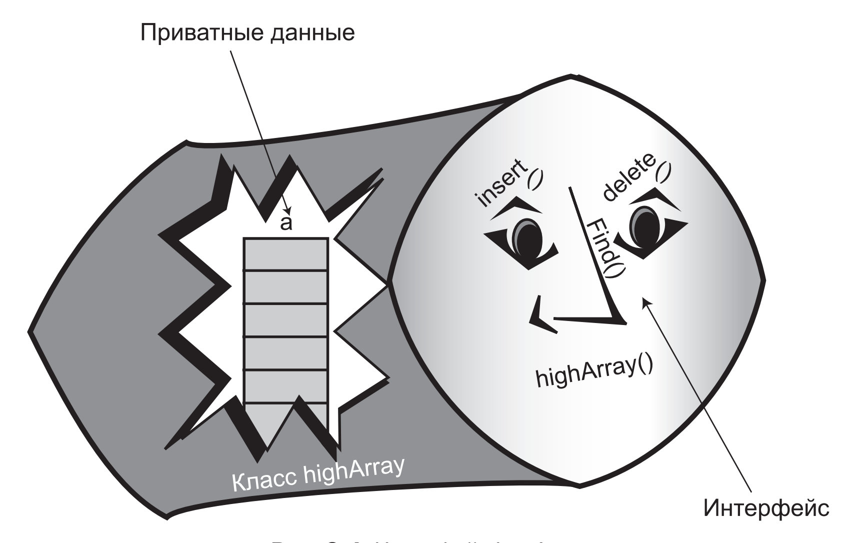


Рис. 2.4. Интерфейс LowArray

Листинг 2.3. Программа highArray.java

```
// highArray.java
// Класс массива с высокоуровневым интерфейсом
// Запуск программы: C>java HighArrayApp
////////////////////////////////////////////////////////////////
class HighArray
   {
   private long[] a;                 // Ссылка на массив a
   private int nElems;               // Количество элементов в массиве
   //-----------------------------------------------------------
   public HighArray(int max)         // Конструктор
      {
      a = new long[max];             // Создание массива
      nElems = 0;                    // Пока нет ни одного элемента
      }
   //-----------------------------------------------------------
```

```
   public boolean find(long searchKey)
      {                              // Поиск заданного значения
      int j;
      for(j=0; j<nElems; j++)        // Для каждого элемента
         if(a[j] == searchKey)       // Значение найдено?
            break;                   // Да - выход из цикла
      if(j == nElems)                // Достигнут последний элемент?
         return false;               // Да
      else
         return true;                // Нет
      }
   //-----------------------------------------------------------
   public void insert(long value)    // Вставка элемента в массив
      {
      a[nElems] = value;             // Собственно вставка
      nElems++;                      // Увеличение размера
   }
   //-----------------------------------------------------------
   public boolean delete(long value)
      {
      int j;
      for(j=0; j<nElems; j++)        // Поиск заданного значения
         if( value == a[j] )
            break;
      if(j==nElems)                  // Найти не удалось
         return false;
      else                           // Значение найдено
         {
         for(int k=j; k<nElems; k++) // Сдвиг последующих элементов
            a[k] = a[k+1];
         nElems--;                   // Уменьшение размера
         return true;
         }
      }
   //-----------------------------------------------------------
   public void display()             // Вывод содержимого массива
      {
         for(int j=0; j<nElems; j++)    // Для каждого элемента
         System.out.print(a[j] + " ");  // Вывод
         System.out.println("");
      }
   //-----------------------------------------------------------
   }  // Конец класса HighArray
////////////////////////////////////////////////////////////////
class HighArrayApp
   {
   public static void main(String[] args)
      {
      int maxSize = 100;            // Размер массива
```

продолжение ⇨

Листинг 2.3 (*продолжение*)

```
      HighArray arr;               // Ссылка на массив
      arr = new HighArray(maxSize); // Создание массива

      arr.insert(77);              // Вставка 10 элементов
      arr.insert(99);
      arr.insert(44);
      arr.insert(55);
      arr.insert(22);
      arr.insert(88);
      arr.insert(11);
      arr.insert(00);
      arr.insert(66);
      arr.insert(33);

      arr.display();               // Вывод элементов

      int searchKey = 35;          // Поиск элемента
      if( arr.find(searchKey) )
         System.out.println("Found " + searchKey);
      else
         System.out.println("Can't find " + searchKey);

      arr.delete(00);              // Удаление трех элементов
      arr.delete(55);
      arr.delete(99);

      arr.display();               // Повторный вывод
      }
   }  // Конец класса HighArrayApp
////////////////////////////////////////////////////////////////
```

Класс HighArray инкапсулирует операции с массивом. Метод main() создает объект этого класса и выполняет практически те же операции, что и программа lowArray.java: вставка 10 элементов, поиск элемента (отсутствующего в массиве) и вывод содержимого массива. Удаление выполняется очень просто, поэтому мы удаляем три элемента (0, 55 и 99) вместо одного и снова выводим содержимое массив. Результат работы программы:

```
77 99 44 55 22 88 11 0 66 33
Can't find 35
77 44 22 88 11 66 33
```

Обратите внимание, каким простым и коротким стал метод main(). Низкоуровневые подробности, которыми раньше приходилось заниматься в методе main() программы lowArray.java, теперь реализуются методами класса HighArray.

Метод find() класса HighArray перебирает массив в поисках элемента с ключевым значением, переданным в аргументе. Метод возвращает true или false в зависимости от того, удалось найти элемент или нет.

Метод insert() помещает новый элемент данных в следующую свободную ячейку массива. Количество ячеек, заполненных данными, хранится в поле

nElems. Методу main() уже не нужно проверять текущее количество элементов массива.

Метод delete() ищет элемент, ключевое значение которого было передано в аргументе. Обнаружив удаляемый элемент, он сдвигает все элементы с большими значениями индексов на одну позицию к началу массива; таким образом, содержимое ячейки с удаленным элементом перезаписывается. Затем метод уменьшает на единицу поле nElems.

В класс также включен метод display() для вывода всех значений, хранящихся в массиве.

Удобство пользователя

В программе lowArray.java (см. листинг 2.2) код поиска элемента в методе main() состоял из восьми строк; в программе highArray.java достаточно одной строки. Пользователю класса (HighArrayApp) не нужно беспокоиться об индексах или других подробностях реализации массива. Более того, пользователю даже не нужно знать, какая структура используется классом HighArray для хранения данных. Структура данных скрыта за интерфейсом. В следующем разделе будет приведен пример использования того же интерфейса с другой структурой данных.

Абстракция

Процесс отделения «как» от «что» (то есть конкретного способа выполнения операции внутри класса от внешнего интерфейса, видимого пользователю класса) называется *абстракцией*. Абстракция принадлежит к числу важных аспектов программотехники. Абстрагирование функциональности класса упрощает проектирование программы, поскольку нам не приходится беспокоиться о подробностях реализации на ранней стадии процесса проектирования.

Приложение Ordered Workshop

Представьте себе массив, в котором элементы данных отсортированы по возрастанию ключевых значений — элемент с минимальным значением имеет индекс 0, а в каждой последующей ячейке ключевое значение больше, чем в предыдущей. Такие массивы называются *упорядоченными*.

При вставке нового элемента в упорядоченный массив необходимо найти для него подходящую позицию: за ближайшим меньшим значением и перед ближайшим большим. Далее все большие значения сдвигаются к концу массива, освобождая место для нового элемента.

Для чего хранить упорядоченные данные? Одно из преимуществ такого способа хранения — радикальное ускорение поиска за счет применения *двоичного поиска*.

Запустите приложение Ordered Workshop (см. главу 1). В окне отображается содержимое массива (рис. 2.5); это приложение похоже на Array Workshop, но данные в нем упорядочены.

Для упорядоченного массива мы запретили дубликаты. Как было показано ранее, это решение ускоряет поиск, но несколько замедляет вставку.

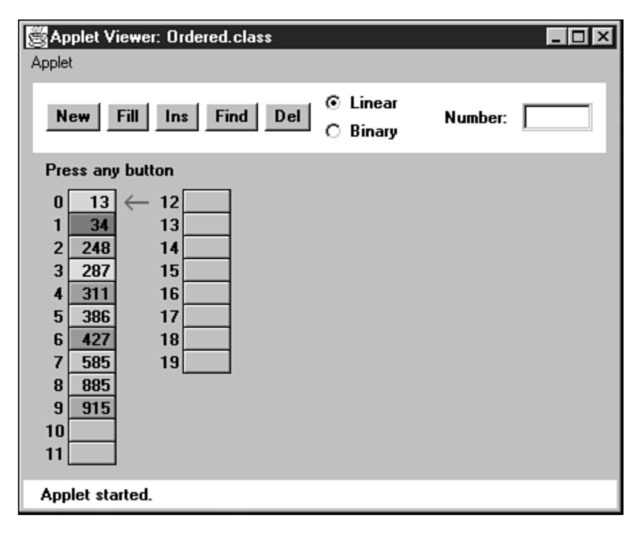


Рис. 2.5. Приложение Ordered Workshop

Линейный поиск

Приложение Ordered Workshop поддерживает два алгоритма поиска: линейный и двоичный. Линейный поиск используется по умолчанию. Он работает по тому же принципу, что и поиск в неупорядоченном массиве в приложении Array: красная стрелка последовательно переходит от элемента к элементу в поисках совпадения. Единственное отличие заключается в том, что при обнаружении элемента с большим ключом поиск прерывается.

Опробуйте процедуру линейного поиска. Убедитесь в том, что в приложении установлен переключатель `Linear`, а затем воспользуйтесь кнопкой `Find` для поиска несуществующего значения, которое, если бы оно присутствовало, находилось бы в середине массива. Например, на рис. 2.5 это могло быть число 400. Вы увидите, что поиск завершается при достижении первого элемента с ключом, превышающим 400; на рисунке это элемент 427. Алгоритм знает, что дальнейший поиск не имеет смысла.

Также опробуйте кнопки `Ins` и `Del`. Используйте кнопку `Ins` для вставки элемента с ключом, значение которого находится где-то в середине диапазона существующих ключей. Вы увидите, что вставка требует перемещения всех элементов с ключами, превышающими вставляемый.

Удалите элемент из середины массива при помощи кнопки `Del`. Удаление работает практически так же, как в приложении Array — элементы с большими индексами сдвигаются к началу массива, чтобы заполнить «дыру», оставшуюся после удаления. В упорядоченном массиве алгоритм удаления может прервать работу на середине, если ему не удастся найти удаляемый элемент (по аналогии с поиском).

Двоичный поиск

Достоинства упорядоченного массива проявляются при использовании двоичного поиска. Такой поиск выполняется намного быстрее линейного, особенно в больших массивах.

Игра «угадай число»

Двоичный поиск основан на том же принципе, что и детская игра «угадай число». В этой игре один игрок загадывает число от 1 до 100, а другой пытается угадать его. Он называет число, а загадавший сообщает: названное число больше загаданного, меньше его или угадано верно.

Чтобы определить число за минимальное количество предположений, всегда следует начинать с 50. Если загаданное число больше названного, значит, оно находится в диапазоне от 51 до 100, и следующим называется число 75 (середина диапазона от 51 до 100). В противном случае загаданное число находится в диапазоне от 1 до 49, и следующим называется число 25.

Каждое предположение сокращает диапазон возможных значений вдвое. Когда в диапазоне остается всего одно число, это и есть правильный ответ.

Обратите внимание на скорость поиска. Если бы мы воспользовались линейным поиском, назвав сначала 1, потом 2, потом 3 и т. д., то для нахождения загаданного числа потребовалось бы в среднем 50 предположений. При двоичном поиске каждое предположение сокращает диапазон вдвое, поэтому для получения ответа требуется намного меньше предположений. В таблице 2.2 показано, как отгадывается число 33.

Таблица 2.2. Игра «Угадай число»

Попытка	Предположение	Результат	Диапазон возможных значений
0			1–100
1	50	Меньше	1–49
2	25	Больше	26–49
3	37	Меньше	26–36
4	31	Больше	32–36
5	34	Меньше	32–33
6	32	Больше	33–33
7	33	Верно	

Правильное число было определено всего за семь попыток, и это максимум — возможно, вам повезет, и вы угадаете число еще до того, как размер диапазона сократится до 1. Например, это произойдет, если загадать число 50 или 34.

Двоичный поиск в приложении Ordered Workshop

Чтобы выполнить двоичный поиск в приложении Ordered Workshop, создайте новый массив (кнопка New). После первого нажатия вам будет предложено задать размер массива (максимум 60) и выбрать алгоритм поиска: линейный или двоичный. Выберите двоичный поиск (переключатель Binary). После того как массив будет создан, заполните его данными при помощи кнопки Fill. Введите количество элементов (не более размера массива).

Когда массив будет заполнен, выберите одно из значений в массиве и понаблюдайте за тем, как происходит его поиск. После нескольких исходных нажатий красная стрелка перемещается к текущему предположению алгоритма, а диапазон возможных значений обозначается вертикальной синей чертой рядом с соответствующими ячейками. На рис. 2.6 показано окно приложения в ситуации, когда диапазон состоит из всех элементов массива.

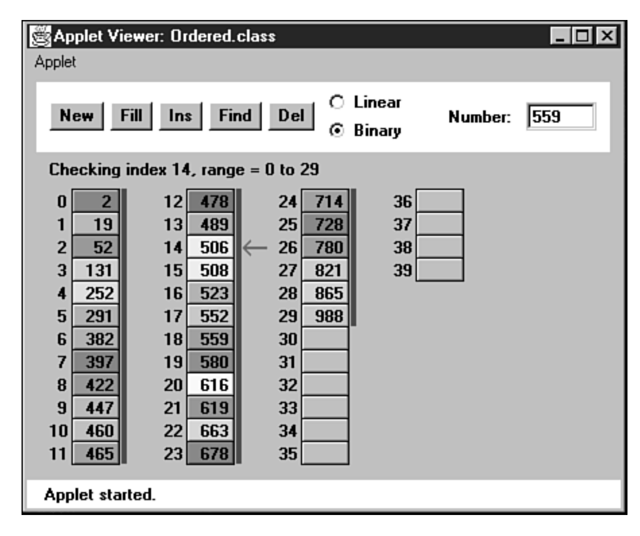


Рис. 2.6. Исходный диапазон при двоичном поиске

При каждом нажатии кнопки Find диапазон сокращается вдвое, а новое предположение выбирается в середине диапазона. На рис. 2.7 показан следующий шаг поиска.

Даже при максимальном размере массива из 60 элементов полудюжины нажатий кнопки хватит для успешного обнаружения любого элемента.

Попробуйте использовать двоичный поиск с массивами разных размеров. Сможете ли вы рассчитать максимальное количество шагов до запуска приложения? Мы вернемся к этому вопросу в последнем разделе этой главы.

Обратите внимание: операции вставки и удаления также поддерживают двоичный поиск. Место вставки элемента определяется двоичным поиском, как и позиция удаляемого элемента. В этом приложении элементы с одинаковыми ключами запрещены.

Рис. 2.7. Диапазон возможных значений на шаге 2 двоичного поиска

Реализация упорядоченного массива на языке Java

Рассмотрим код реализации упорядоченного массива на языке Java. Для инкапсуляции массива и его алгоритмов будет использован класс OrdArray. Центральное место в этом классе занимает метод find(), который использует двоичный поиск для обнаружения заданного элемента данных. Мы подробно рассмотрим этот метод, прежде чем переходить к полному коду программы.

Двоичный поиск с методом find()

Метод find() ищет заданный элемент многократным половинным делением диапазона элементов массива. Метод выглядит так:

```
public int find(long searchKey)
   {
   int lowerBound = 0;
   int upperBound = nElems-1;
   int curIn;

   while(true)
      {
      curIn = (lowerBound + upperBound ) / 2;
      if(a[curIn]==searchKey)
         return curIn;              // Элемент найден
```

```
      else if(lowerBound > upperBound)
         return nElems;              // Элемент не найден
      else                           // Деление диапазона
         {
         if(a[curIn] < searchKey)
            lowerBound = curIn + 1; // В верхней половине
         else
            upperBound = curIn - 1; // В нижней половине
         }
      }
   }
```

В самом начале метода переменным lowerBound и upperBound приисваиваются индексы первой и последней занятой ячейки массива. Эти переменные определяют диапазон, в котором может находиться элемент с искомым ключом searchKey. Затем в цикле while текущий индекс curIn устанавливается в середину этого диапазона.

Если повезет, curIn сразу укажет на искомый элемент, поэтому мы сначала проверяем эту возможность. Если элемент найден, мы возвращаем его индекс curIn. При каждой итерации цикла диапазон уменьшается вдвое. В конечном итоге он сократится настолько, что дальнейшее деление станет невозможным. Эта ситуация проверяется в следующей команде: если lowerBound больше upperBound, значит, поиск завершен. (Если индексы равны, то диапазон состоит из одного элемента, и понадобится еще одна итерация цикла.) Поиск не может продолжаться без диапазона; если заданный элемент не найден, метод возвращает nElems, общее количество элементов. Это значение не является действительным индексом, потому что последняя заполненная ячейка массива имеет индекс nElems-1. Пользователь класса интерпретирует это значение как признак того, что элемент не был найден.

Если curIn не указывает на искомый элемент, а диапазон все еще остается достаточно большим, то его следует разделить пополам. Мы сравниваем значение по текущему индексу a[curIn], находящееся в середине диапазона, с искомым значением searchKey.

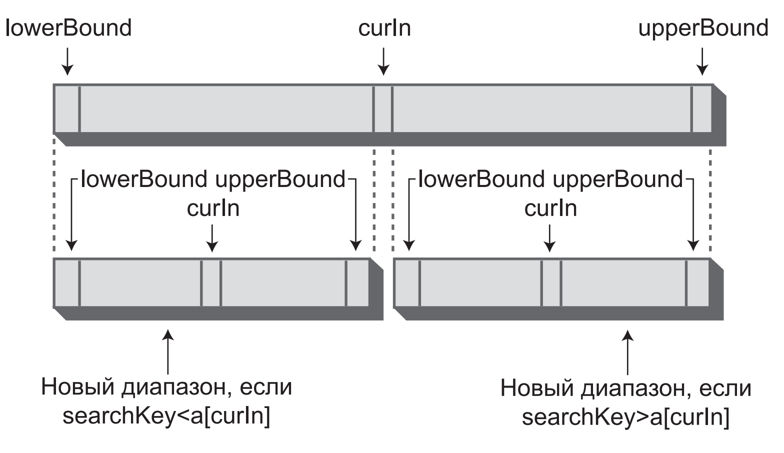


Рис. 2.8. Деление диапазона при двоичном поиске

Если значение searchKey больше, значит, поиск должен осуществляться в верхней половине диапазона. Соответственно lowerBound нужно присвоить значение curIn, а точнее индекс следующей ячейки, потому что ячейка curIn уже была проверена в начале цикла.

Если значение searchKey меньше a[curIn], то поиск следует ограничить нижней половиной диапазона. Соответственно upperBound смещается до ячейки, предшествующей curIn. На рис. 2.8 показано, как изменяется диапазон в этих двух ситуациях.

Класс OrdArray

В целом программа orderedArray.java похожа на highArray.java (листинг 2.3). Главное отличие заключается в том, что find() использует двоичный поиск.

Двоичный поиск можно было бы использовать и для поиска позиции вставки нового элемента. Тем не менее для простоты в insert() был сохранен линейный поиск. Снижение скорости в этом случае не столь существенно, потому что, как было показано ранее, при выполнении вставки все равно приходится перемещать около половины элементов, поэтому вставка будет относительно медленной операцией даже при использовании двоичного поиска. Но если вам все же потребуется «выжать» максимальную скорость выполнения операции, запрограммируйте в начальной части insert() двоичный поиск (как было сделано в приложении Ordered Workshop). Кроме того, метод delete() может вызвать find() для определения позиции удаляемого элемента.

Класс OrdArray содержит новый метод size(), который возвращает текущее количество элементов данных в массиве. Эта информация важна для пользователя класса main() при вызове find(). Если If find() вернет значение nElems, которое main() идентифицирует при помощи size(), значит, поиск был безуспешным. В листинге 2.4 приведен полный текст программы orderedArray.java.

Листинг 2.4. Программа orderedArray.java

```
// orderedArray.java
// Работа с классом упорядоченного массива
// Запуск программы: C>java OrderedApp
////////////////////////////////////////////////////////////////
class OrdArray
   {
   private long[] a;                 // Ссылка на массив a
   private int nElems;               // Количество элементов данных
   //-----------------------------------------------------------
   public OrdArray(int max)          // Конструктор
      {
      a = new long[max];             // Создание массива
      nElems = 0;
      }
   //-----------------------------------------------------------
```

продолжение ⇨

Листинг 2.4 (*продолжение*)

```
   public int size()
      { return nElems; }
   //-----------------------------------------------------------
   public int find(long searchKey)
      {
      int lowerBound = 0;
      int upperBound = nElems-1;
      int curIn;

      while(true)
         {
         curIn = (lowerBound + upperBound ) / 2;
         if(a[curIn]==searchKey)
            return curIn;              // Элемент найден
         else if(lowerBound > upperBound)
            return nElems;             // Элемент не найден
         else                          // Деление диапазона
            {
            if(a[curIn] < searchKey)
               lowerBound = curIn + 1; // В верхней половине
            else
               upperBound = curIn - 1; // В нижней половине
            }
         }
      }
   //-----------------------------------------------------------
   public void insert(long value)    // Вставка элемента в массив
      {
      int j;
      for(j=0; j<nElems; j++)        // Определение позиции вставки
         if(a[j] > value)            // (линейный поиск)
            break;
      for(int k=nElems; k>j; k--)    // Перемещение последующих элементов
         a[k] = a[k-1];
      a[j] = value;                  // Вставка
      nElems++;                      // Увеличение размера
      }
   //-----------------------------------------------------------
   public boolean delete(long value)
      {
      int j = find(value);
      if(j==nElems)                  // Найти не удалось
         return false;
      else                           // Элемент найден
         {
         for(int k=j; k<nElems; k++) // Перемещение последующих элементов
            a[k] = a[k+1];
         nElems--;                   // Уменьшение размера
```

```
         return true;
      }
   }
   //-----------------------------------------------------------
   public void display()            // Вывод содержимого массива
      {
      for(int j=0; j<nElems; j++)       // Перебор всех элементов
         System.out.print(a[j] + " ");  // Вывод текущего элемента
      System.out.println("");
      }
   //-----------------------------------------------------------
   }
////////////////////////////////////////////////////////////////
class OrderedApp
   {
   public static void main(String[] args)
      {
      int maxSize = 100;             // Размер массива
      OrdArray arr;                  // Ссылка на массив
      arr = new OrdArray(maxSize);   // Создание массива

      arr.insert(77);                // Вставка 10 элементов
      arr.insert(99);
      arr.insert(44);
      arr.insert(55);
      arr.insert(22);
      arr.insert(88);
      arr.insert(11);
      arr.insert(00);
      arr.insert(66);
      arr.insert(33);

      int searchKey = 55;            // Поиск элемента
      if( arr.find(searchKey) != arr.size() )
         System.out.println("Found " + searchKey);
      else
         System.out.println("Can't find " + searchKey);

      arr.display();                 // Вывод содержимого

      arr.delete(00);                // Удаление трех элементов
      arr.delete(55);
      arr.delete(99);

      arr.display();                 // Повторный вывод
      }
   }  Конец класса OrderedApp
////////////////////////////////////////////////////////////////
```

Преимущества упорядоченных массивов

Что же нам дало использование упорядоченного массива? Главное преимущество — значительное ускорение поиска по сравнению с неупорядоченным массивом. С другой стороны, вставка стала занимать больше времени, потому что нам приходится сдвигать все элементы данных с большими значениями ключа для освобождения места. Удаление выполняется медленно как в упорядоченных, так и неупорядоченных массивах, так как элементы приходится сдвигать к концу массива для заполнения «дыры», оставшейся от удаления.

Таким образом, упорядоченные массивы удобны в ситуациях, когда поиск выполняется часто, а операции вставки и удаления относительно редки. Например, он хорошо подойдет для базы данных работников компании. Наем новых работников и увольнение выполняются относительно редко по сравнению с обращениями к записям существующих работников за информацией или их обновлением при изменении адреса, зарплаты и т. д.

Напротив, упорядоченный массив вряд ли подойдет для складской базы розничного магазина — часто выполняемые операции вставки и удаления по мере поступления и продажи товаров будут выполняться медленно.

Логарифмы

В этом разделе будет показано, как использовать логарифмы для вычисления количества необходимых шагов при двоичном поиске. Читатели, еще не забывшие школьный курс математики, вероятно, могут пропустить его. Читатели, у которых математические вычисления вызывают аллергию, могут сделать то же самое — им достаточно хорошенько изучить табл. 2.3.

Мы видели, что двоичный поиск обладает значительными преимуществами перед линейным поиском. В игре «угадай число» с диапазоном от 1 до 100 для угадывания любого числа методом двоичного поиска требуется не более семи попыток; точно так же в массиве из 100 элементов запись с заданным ключом находится за семь сравнений. Как насчет других диапазонов? В таблице 2.3 приведены примеры диапазонов и количество сравнений, необходимых для двоичного поиска.

Обратите внимание на различия между скоростью двоичного и линейного поиска. При малом количестве элементов различия не столь существенны. Поиск в 10 элементах требует 5 сравнений при линейном поиске ($N/2$) и максимум 4 сравнения при двоичном поиске. Однако чем больше элементов в массиве, тем заметнее становится разница. Со 100 элементами линейный поиск требует 50 сравнений, а двоичный — только 7. С 1000 элементов разрыв увеличивается до соотношения 500/10, а с 1 000 000 элементов — до соотношения 500 000/20. Можно сделать вывод, что для всех массивов, кроме самых малых, двоичный поиск обеспечивает заметный выигрыш по скорости.

Таблица 2.3. Количество необходимых сравнений при двоичном поиске

Диапазон	Необходимо сравнений
10	4
100	7
1 000	10
10 000	14
100 000	17
1 000 000	20
10 000 000	24
100 000 000	27
1 000 000 000	30

Формула

Результаты из табл. 2.3 легко проверить: последовательно делите диапазон (из первого столбца) пополам, пока он не станет слишком малым для дальнейшего деления. Количество необходимых делений в этом процессе определяет количество сравнений из второго столбца.

Многократное деление диапазона надвое — алгоритмический способ определения количества сравнений. Нельзя ли получить нужное число по простой формуле? Конечно, такая формула существует, и ее стоит рассмотреть здесь, потому что она время от времени встречается при изучении эффективности структур данных. Формула основана на использовании логарифмов (без паники!).

Числа в табл. 2.3 не дают ответа на некоторые вопросы: например, каков точный размер массива, поиск в котором заведомо может быть завершен за пять шагов? Для решения этой проблемы необходимо создать похожую таблицу, которая начинает с одного элемента и последовательно увеличивает его вдвое. В табл. 2.4 приведены данные для первых семи шагов.

В исходной ситуации с массивом из 100 элементов мы видим, что 6 шагов дают недостаточно большой диапазон (64), а 7 шагов его перекрывают с избытком (128). Таким образом, 7 шагов для 100 элементов в табл. 2.3 верны, как и 10 шагов для 1000 элементов.

Последовательное удвоение размера диапазона создает числовой ряд, члены которого равны двум в соответствующей степени (третий столбец табл. 2.4). Если s — количество шагов (количество умножений на 2, то есть степень, в которую возводится 2), а r — размер диапазона, то формула выглядит так:

$$r = 2^s$$

Если известно значение s (количество шагов), то по нему можно определить величину диапазона. Например, если $s = 6$, то диапазон составляет 2^6, или 64.

Таблица 2.4. Степени двойки

Шаг s (то же, что $\log_2(r)$)	Диапазон r	Диапазон, выраженный в виде степени 2 (2^s)
0	1	2^0
1	2	2^1
2	4	2^2
3	8	2^3
4	16	2^4
5	32	2^5
6	64	2^6
7	128	2^7
8	256	2^8
9	512	2^0
10	1024	2^{10}

Операция, обратная возведению в степень

Наша исходная задача была противоположна только что приведенной: размер диапазона известен, требуется определить, сколько сравнений понадобится для завершения поиска. Иначе говоря, по известному *r* требуется определить *s*.

Операция, обратная возведению в степень, называется логарифмом. Вот как выглядит нужная формула с логарифмом:

$$s = \log_2(r).$$

Из формулы следует, что количество шагов (сравнений) равно логарифму размера диапазона по основанию 2. Что это значит? Логарифмом числа *r* по основанию 2 называется степень, в которую необходимо возвести 2 для получения *r*. В таблице 2.4 числа в первом столбце *s* равны $\log_2(r)$.

Как узнать логарифм числа без многочисленных делений? На карманных калькуляторах и во многих компьютерных языках существует функция log. Обычно логарифм вычисляет по основанию 10, но результат легко преобразуется к основанию 2 — достаточно умножить его на 3,322. Например, $\log_{10}(100) = 2$; следовательно, $\log_2(100) = 2 \times 3{,}322$, или 6,644. При округлении вверх до целого числа 7 мы получаем значение в столбце справа от 100 в табл. 2.4.

Как бы то ни было, мы не будем заниматься вычислением конкретных логарифмов. Важнее понять связь между числом и его логарифмом. Еще раз взгляните на табл. 2.3, сравнивающую количество элементов с количеством шагов, необходимых для поиска конкретного элемента. При 10-кратном увеличении количество элементов (размера диапазона) число шагов возрастает всего на 3 или 4 (а точнее 3,322 — перед округлением вверх). Скорость роста логарифма значительно отстает от скорости роста самого числа. Мы сравним логарифмическую скорость роста с другими математическими функциями при обсуждении «O-синтаксиса».

Хранение объектов

Во всех примерах на языке Java, приводившихся до настоящего момента, в структурах данных хранились примитивные переменные типа `long`. Хранение примитивных типов упрощает примеры, но оно не характерно для использования реальных структур данных. Обычно хранимые элементы данных (записи) состоят из многих полей. Скажем, в записи работника может храниться фамилия, имя, возраст, номер социального страхования и т. д. В каталоге коллекции марок хранится название страны, выпустившей марку, номер по каталогу, состояние, текущая стоимость и т. д.

Следующий пример демонстрирует хранение объектов вместо переменных примитивных типов.

Класс Person

В Java запись данных обычно представляется объектом класса. Возьмем класс `Person` — типичный класс, используемый для хранения данных:

```
class Person
   {
   private String lastName;
   private String firstName;
   private int age;
   //-------------------------------------------------------------
   public Person(String last, String first, int a)
      {                                // Конструктор
      lastName = last;
      firstName = first;
      age = a;
      }
   //-------------------------------------------------------------
   public void displayPerson()
      {
      System.out.print("   Last name: " + lastName);
      System.out.print(", First name: " + firstName);
      System.out.println(", Age: " + age);
      }
   //-------------------------------------------------------------
   public String getLast()           // Получение фамилии
      { return lastName; }
   }  // Конец класса Person
```

Класс содержит всего три переменные: имя, фамилия и возраст человека. Конечно, в реальных приложениях записи могут содержать много дополнительных полей. Конструктор позволяет создать новый объект `Person` с инициализацией его полей. Метод `displayPerson()` выводит данные объекта `Person`, а метод `getLast()` возвращает значение `lastName` — ключевого поля, используемого при поиске.

Программа classDataArray.java

Программа, использующая класс Person, напоминает программу highArray.java (листинг 2.3), в которой хранились элементы типа long. Чтобы перевести программу на хранение объектов Person, достаточно внести несколько изменений; основные из них:

- Тип массива a изменяется на Person.
- Ключевое поле (lastName) теперь является объектом String, поэтому в сравнениях должен использоваться метод equals() вместо оператора ==. Метод getLast() класса Person возвращает значение lastName объекта Person, а метод equals() выполняет сравнение:

```
if( a[j].getLast().equals(searchName) ) // Элемент найден?
```

- Метод insert() создает новый объект Person и вставляет его в массив (вместо вставки значения long).

Метод main() слегка изменен — в основном из-за усложнившегося вывода. Программа по-прежнему вставляет 10 элементов, выводит их, ищет один элемент, удаляет три элемента и снова выводит их. Полный код программы classDataArray.java приведен в листинге 2.5.

Листинг 2.5. Программа classDataArray.java

```
// classDataArray.java
// Объекты как элементы данных
// Запуск программы: C>java ClassDataApp
////////////////////////////////////////////////////////////////
class Person
   {
   private String lastName;
   private String firstName;
   private int age;
//--------------------------------------------------------------
   public Person(String last, String first, int a)
      {                               // Конструктор
      lastName = last;
      firstName = first;
      age = a;
      }
//--------------------------------------------------------------
   public void displayPerson()
      {
      System.out.print("   Last name: " + lastName);
      System.out.print(", First name: " + firstName);
      System.out.println(", Age: " + age);
      }
//--------------------------------------------------------------
   public String getLast()           // Получение фамилии
      { return lastName; }
   }  // Конец класса Person
```

```
//////////////////////////////////////////////////////////////////
class ClassDataArray
   {
   private Person[] a;              // Ссылка на массив
   private int nElems;              // Количество элементов данных

   public ClassDataArray(int max)   // Конструктор
      {
      a = new Person[max];              // Создание массива
      nElems = 0;                       // Пока нет ни одного элемента
      }
//-----------------------------------------------------------------
   public Person find(String searchName)
      {                                 // Поиск заданного значения
      int j;
      for(j=0; j<nElems; j++)           // Для каждого элемента
         if( a[j].getLast().equals(searchName) )  // Значение найдено?
            break;                      // Выход из цикла
      if(j == nElems)                   // Достигнут последний элемент?
         return null;                   // Да, значение не найдено
      else
         return a[j];                   // Нет, значение найдено
      }  // end find()
//----------------------------------------------------------------
   // Включение записи в массив
   public void insert(String last, String first, int age)
      {
      a[nElems] = new Person(last, first, age);
      nElems++;                         // Увеличение размера
      }
//-----------------------------------------------------------------
   public boolean delete(String searchName)
      {                                 // Удаление из массива
      int j;
      for(j=0; j<nElems; j++)           // Поиск удаляемого элемента
         if( a[j].getLast().equals(searchName) )
            break;
      if(j==nElems)                     // Найти не удалось
         return false;
      else                              // Значение найдено
         {
         for(int k=j; k<nElems; k++)    // Сдвиг последующих элементов
            a[k] = a[k+1];
         nElems--;                      // Уменьшение размера
         return true;
         }
      }
```

продолжение ⇒

Листинг 2.5 (*продолжение*)

```
//--------------------------------------------------------------
   public void displayA()               // Вывод содержимого массива
      {
      for(int j=0; j<nElems; j++)       // Для каждого элемента
         a[j].displayPerson();          // Вывод
      }
//--------------------------------------------------------------
   }  // Конец класса ClassDataArray
////////////////////////////////////////////////////////////////
class ClassDataApp
   {
   public static void main(String[] args)
      {
      int maxSize = 100;             // Размер массива
      ClassDataArray arr;            // Ссылка на массив
      arr = new ClassDataArray(maxSize);  // Создание массива
                                     // Вставка 10 элементов
      arr.insert("Evans", "Patty", 24);
      arr.insert("Smith", "Lorraine", 37);
      arr.insert("Yee", "Tom", 43);
      arr.insert("Adams", "Henry", 63);
      arr.insert("Hashimoto", "Sato", 21);
      arr.insert("Stimson", "Henry", 29);
      arr.insert("Velasquez", "Jose", 72);
      arr.insert("Lamarque", "Henry", 54);
      arr.insert("Vang", "Minh", 22);
      arr.insert("Creswell", "Lucinda", 18);

      arr.displayA();                // Вывод содержимого

      String searchKey = "Stimson";  // Поиск элемента
      Person found;
      found=arr.find(searchKey);
      if (found != null)
         {
         System.out.print("Found ");
         found.displayPerson();
         }
      else
         System.out.println("Can't find " + searchKey);

      System.out.println("Deleting Smith, Yee, and Creswell");
      arr.delete("Smith");           // Удаление трех элементов
      arr.delete("Yee");
      arr.delete("Creswell");

      arr.displayA();                // Повторный вывод содержимого
      }
   }  // Конец класса ClassDataApp
////////////////////////////////////////////////////////////////
```

Результат выполнения программы:

```
Last name: Evans, First name: Patty, Age: 24
Last name: Smith, First name: Lorraine, Age: 37
Last name: Yee, First name: Tom, Age: 43
Last name: Adams, First name: Henry, Age: 63
Last name: Hashimoto, First name: Sato, Age: 21
Last name: Stimson, First name: Henry, Age: 29
Last name: Velasquez, First name: Jose, Age: 72
Last name: Lamarque, First name: Henry, Age: 54
Last name: Vang, First name: Minh, Age: 22
Last name: Creswell, First name: Lucinda, Age: 18
Found    Last name: Stimson, First name: Henry, Age: 29
Deleting Smith, Yee, and Creswell
Last name: Evans, First name: Patty, Age: 24
Last name: Adams, First name: Henry, Age: 63
Last name: Hashimoto, First name: Sato, Age: 21
Last name: Stimson, First name: Henry, Age: 29
Last name: Velasquez, First name: Jose, Age: 72
Last name: Lamarque, First name: Henry, Age: 54
Last name: Vang, First name: Minh, Age: 22
```

Программа `classDataArray.java` наглядно показывает, что структуры данных могут использоваться не только для хранения примитивных типов, но и объектов классов. (Обратите внимание: реальная программа, использующая фамилию в качестве ключа, должна принять во внимание возможность дублирования фамилий — как упоминалось ранее, это несколько усложнит программирование.)

О-синтаксис

В США автомобили делятся в зависимости от размера на несколько категорий: субкомпакты, компакты, среднеразмерные и т. д. Эти категории позволяют получить общее представление о габаритах автомобиля без указания конкретных размеров. Подобная система приблизительных обозначений существует и для описания эффективности компьютерных алгоритмов. В информатике она называется «О-синтаксисом».

На первый взгляд может показаться, что сравнение алгоритмов должно базироваться на оценках вида «Алгоритм А работает вдвое быстрее алгоритма Б», но в действительности такие утверждения не имеют особого смысла. Почему? Потому что пропорции могут кардинально изменяться при изменении количества элементов. Допустим, при увеличении количества элементов на 50% алгоритм А начинает работать *втрое* быстрее алгоритма Б, а при половинном количестве элементов алгоритмы А и Б сравниваются по скорости. Критерий сравнения (*сложность алгоритма*) должен связывать скорость алгоритма с количеством элементов. Давайте проанализируем эту связь для алгоритмов, рассмотренных ранее.

Вставка в неупорядоченный массив: постоянная сложность

Вставка в неупорядоченный массив — единственный из рассмотренных нами алгоритмов, не зависящих от количества элементов в массиве. Новый элемент всегда размещается в следующей свободной ячейке `a[nElems]`, после чего значение `nElems` увеличивается. Вставка всегда выполняется с постоянной скоростью независимо от N — количества элементов в массиве. Можно сказать, что время вставки элемента в неупорядоченный массив T является константой K:

$$T = K.$$

В реальной ситуации фактическое время вставки (в микросекундах или других единицах) зависит от скорости микропроцессора, эффективности сгенерированного компилятором кода и других факторов. Константа K в приведенной формуле учитывает все эти факторы. Чтобы узнать конкретное значение K в реальной ситуации, необходимо измерить время выполнения операции. (Для этой цели существуют специальные программы.)

Линейный поиск: сложность пропорциональна N

Мы уже видели, что при линейном поиске элементов в массиве количество сравнений для поиска заданного значения в среднем составляет половину от общего количества элементов. Таким образом, если N — общее количество элементов, то время поиска T пропорционально половине N:

$$T = K \times N/2.$$

Как и в случае вставки, вычисление константы K в этой формуле потребует поиска по некоторому (вероятно, большому) значению N. Далее по измеренному результату T определяется константа K, которая в дальнейшем используется для вычисления T по любому другому значению N. Чтобы формула стала более удобной, делитель 2 можно внести в K; новое значение K равно старому, разделенному на 2. Формула приходит к итоговому виду:

$$T = K \times N.$$

Из этой формулы следует, что среднее время линейного поиска пропорционально размеру массива. Если увеличить размер массива вдвое, то поиск займет вдвое больше времени.

Двоичный поиск: сложность пропорциональна log(N)

Аналогичным образом строится формула, связывающая T с N для двоичного поиска:

$$T = K \times \log_2(N).$$

Как было показано ранее, время пропорционально логарифму N по основанию 2. Но так как любой логарифм прямо пропорционален логарифму по другому основанию (коэффициент 3,322 для перехода от двоичного логарифма к десятичному), эту константу тоже можно внести в K; а следовательно, указывать основание логарифма не обязательно:

$$T = K \times \log(N).$$

Константа не нужна

«О-синтаксис» почти не отличается от приведенных формул, если не считать того, что в нем не используется константа K. При сравнении алгоритмов особенности конкретного микропроцессора или компилятора несущественны; важна только закономерность изменения T для разных значений N, а не конкретные числа. А это значит, что постоянный коэффициент не нужен.

В «О-синтаксисе» используется прописная буква «О»; считайте, что он обозначает «порядок» (Order of). В «О-синтаксисе» линейный поиск выполняется за время $O(N)$, а двоичный поиск — за время $O(\log N)$. Вставка в неупорядоченный массив выполняется за время $O(1)$, то есть за постоянное время (обозначаемое константой 1).

В таблице 2.5 приведена сводка формул сложности всех алгоритмов, рассматривавшихся до настоящего момента, записанных в О-синтаксисе.

Таблица 2.5. Время выполнения операций в О-синтаксисе

Алгоритм	Время выполнения в О-синтаксисе
Линейный поиск	$O(N)$
Двоичный поиск	$O(\log N)$
Вставка в неупорядоченном массиве	$O(1)$
Вставка в упорядоченном массиве	$O(N)$
Удаление в неупорядоченном массиве	$O(N)$
Удаление в упорядоченном массиве	$O(N)$

На рис. 2.9 показаны некоторые зависимости между временем выполнения операции и количеством элементов. На основании этого графика можно (весьма условно) оценить разные сложности: $O(1)$ — отлично, $O(\log N)$ — хорошо, $O(N)$ — неплохо, $O(N^2)$ — плохо. Сложность $O(N^2)$ встречается при пузырьковой сортировке и в некоторых алгоритмах работы с графами, которые будут рассматриваться позднее.

О-синтаксис не дает конкретных цифр, а передает общий характер зависимости времени выполнения от количества элементов. Это самый информативный механизм сравнения алгоритмов, не считая разве что замера времени выполнения в реальных условиях.

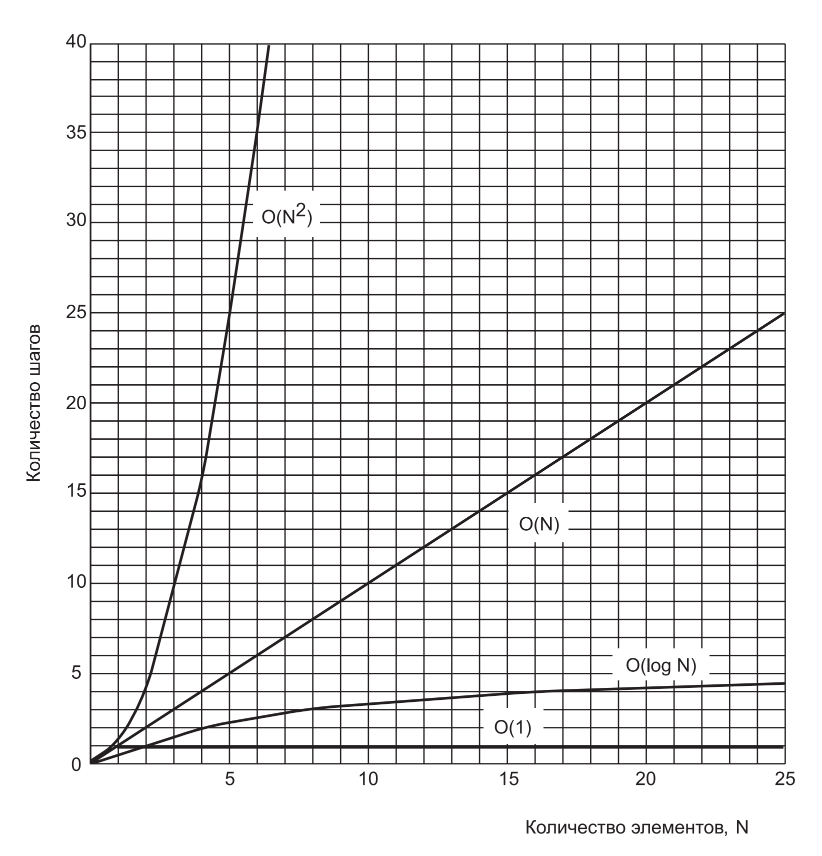


Рис. 2.9. Сложность выполнения различных операций

Почему бы не использовать только массивы?

Массивы поддерживают все необходимые операции, так почему бы не использовать их для всех задач хранения данных? Некоторые недостатки массивов уже были описаны ранее. В неупорядоченном массиве вставка выполняется быстро, за время $O(1)$, зато медленный поиск требует времени $O(N)$. Удаление в обоих видах массивов выполняется за время $O(N)$, потому что для заполнения «дыры» приходится перемещать (в среднем) половину элементов.

Конечно, нам хотелось бы, чтобы структуры данных могли выполнять все операции — вставку, удаление, поиск — быстро, в идеале за время $O(1)$, а если нет — то хотя бы за время $O(\log N)$. В следующих главах вы увидите, насколько близко можно подойти к этому идеалу и как за это придется расплачиваться усложнением алгоритмов.

Другой недостаток массивов заключается в том, что их размер фиксируется при создании оператором `new`. Обычно при запуске программы вы еще не знаете, сколько элементов будет размещено в массиве, поэтому размер массива приходится угадывать. Завышенная оценка приведет к неэффективному расходованию памяти под ячейки массива, которые никогда не заполняются. Если оценка окажется заниженной, то переполнение массива в лучшем случае приведет к сообщению об невозможности вставки, а в худшем — к аварийному завершению программы.

Другие структуры данных обладают большей гибкостью и могут расширяться для хранения добавляемых в них элементов. Примером такой структуры является связанный список, описанный в главе 5, «Связанные списки».

Следует заметить, что в Java существует класс `Vector`, который во многих отношениях сходен с массивами, но поддерживает динамическое расширение. За дополнительные возможности приходится расплачиваться частичной потерей быстродействия.

При желании вы можете создать собственную реализацию класса вектора. Если операция пользователя должна привести к переполнению внутреннего массива этого класса, алгоритм вставки создает новый массив большего размера, копирует в него содержимое старого массива, а затем вставляет новый элемент. Весь процесс расширения должен оставаться невидимым для пользователя.

Итоги

- Массивы в Java являются объектами и создаются оператором `new`.
- В неупорядоченных массивах вставка выполняется быстрее, а поиск и удаление — медленнее.
- Инкапсуляция массива в классе защищает массив от случайных изменений.
- Интерфейс класса состоит из методов (а иногда и полей), с которыми может работать пользователь класса.
- Интерфейс класса часто проектируется так, чтобы упростить выполнение операций для пользователя класса.
- В упорядоченных массивах может применяться двоичный поиск.
- Логарифм числа A по основанию B равен (приблизительно) количеству последовательных делений A на B, пока результат не станет меньше 1.
- Линейный поиск выполняется за время, пропорциональное количеству элементов в массиве.
- Двоичный поиск выполняется за время, пропорциональное логарифму количества элементов.
- O-синтаксис предоставляет удобный способ сравнения скорости алгоритмов.
- Алгоритм, выполняемый за время O(1), является самым эффективным; за время $O(\log N)$ — хорошим, $O(N)$ — неплохим и за время $O(N^2)$ — достаточно плохим.

Вопросы

Следующие вопросы помогут читателю проверить качество усвоения материала. Ответы на них приведены в приложении В.

1. Вставка элемента в неупорядоченный массив:

   a) выполняется за время, пропорциональное размеру массива;

   b) требует нескольких сравнений;

   c) требует сдвига других элементов для освобождения места;

   d) выполняется за постоянное время независимо от количества элементов.

2. При удалении элемента из неупорядоченного массива в большинстве случаев приходится сдвигать другие элементы для заполнения освободившейся ячейки (Да/Нет).

3. В неупорядоченном массиве возможность хранения дубликатов:

   a) увеличивает время выполнения всех операций;

   b) увеличивает время поиска в некоторых ситуациях;

   c) всегда увеличивает время вставки;

   d) иногда увеличивает время вставки.

4. В неупорядоченном массиве проверка отсутствия элемента в массиве обычно занимает меньше времени, чем проверка его присутствия (Да/Нет).

5. При создании массива в Java необходимо использовать ключевое слово _________.

6. Если класс A должен использовать класс B для каких-то целей, то:

   a) методы класса A должны быть простыми для понимания;

   b) желательно, чтобы класс B взаимодействовал с пользователем программы;

   c) более сложные операции следует разместить в классе A;

   d) чем большая часть работы может быть выполнена в классе B, тем лучше.

7. Если класс A использует класс B для каких-то целей, то методы и поля класса B, с которыми может работать класс A, образуют ____________ класса B.

8. В упорядоченных массивах по сравнению с неупорядоченными:

   a) намного быстрее выполняется удаление;

   b) быстрее выполняется вставка;

   c) быстрее выполняется создание;

   d) быстрее выполняется поиск.

9. Логарифм представляет собой операцию, обратную ________________ .

10. Логарифм 1000 по основанию 10 равен ______ .

11. Максимальное количество элементов, которые необходимо проверить для завершения двоичного поиска в массиве из 200 элементов, равно:

    a) 200;

    b) 8;

c) 1;

d) 13.

12. Логарифм 64 по основанию 2 равен _______ .
13. Логарифм 100 по основанию 2 равен 2 (Да/Нет).
14. O-синтаксис описывает:

    a) зависимость сложности алгоритма от количества элементов;

    b) время выполнения алгоритма для структуры данных заданного размера;

    c) время выполнения алгоритма для заданного количества элементов;

    d) зависимость размера структуры данных от количества элементов.
15. Запись $O(1)$ означает, что процесс выполняется за __________ время.
16. В массиве могут храниться как переменные примитивных типов, так и __________.

Упражнения

Упражнения помогут вам глубже усвоить материал этой главы. Программирования они не требуют.

1. Используйте приложение Array Workshop для вставки, поиска и удаления элементов. Убедитесь в том, что вы полностью понимаете, как он работает. Выполните упражнение как с дубликатами, так и без них.
2. Убедитесь в том, что вы можете заранее предсказать, какой диапазон будет выбираться приложением Ordered Workshop на каждом шаге.
3. В массиве с нечетным количеством элементов данных нет среднего элемента. Какой элемент будет сначала проверен алгоритмом двоичного поиска? Проверьте с помощью приложения Ordered Workshop.

Программные проекты

В этом разделе читателю предлагаются программные проекты, которые укрепляют понимание материала и демонстрируют практическое применение концепций этой главы.

2.1. Добавьте в класс `HighArray` из програмы `highArray.java` (листинг 2.3) метод `getMax()`, который возвращает наибольшее значение ключа в массиве или –1, если массив пуст. Добавьте в `main()` код для тестирования нового метода. Считайте, что все ключи являются положительными числами.

2.2. Измените метод из пункта 2.1 так, чтобы элемент с наибольшим ключом не только возвращался методом, но и удалялся из массива. Присвойте новой версии имя `removeMax()`.

2.3. Метод removeMax() из пункта 2.2 может использоваться для сортировки содержимого массива по ключу. Реализуйте алгоритм сортировки, который не изменяет класса HighArray (а изменяет только код main()). Вам потребуется второй массив для хранения отсортированных данных. (Этот алгоритм представляет собой крайне примитивную разновидность сортировки методом выбора, описанной в главе 3, «Простая сортировка».)

2.4. Измените программу orderedArray.java (листинг 2.4) так, чтобы методы insert() и delete(), а также метод find() использовали двоичный поиск (как предлагается в тексте).

2.5. Добавьте в класс OrdArray программы orderedArray.java (листинг 2.4) метод merge(), объединяющий два упорядоченных исходных массива в один упорядоченный приемный массив. Включите в main() фрагмент кода, который заполняет два исходных массива случайными числами, вызывает merge() и выводит содержимое полученного массива. Исходные массивы могут содержать разное количество элементов. Ваш алгоритм должен сравнивать ключи исходных массивов и копировать меньший в приемный массив. Также необходимо предусмотреть ситуацию, когда элементы в одном исходном массиве заканчиваются раньше, чем в другом.

2.6. Добавьте в класс HighArray программы highArray.java (листинг 2.3) метод noDups(), удаляющий все дубликаты из массива. Другими словами, если массив содержит три элемента с ключом 17, метод noDups() должен удалить два из них. Не беспокойтесь о сохранении порядка элементов. Одно из возможных решений — сравнить каждый элемент со всеми остальными элементами и заменить все дубликаты null (или другим значением, не встречающимся среди реальных ключей), после чего удалить из массива все вхождения null. Конечно, размер массива при этом уменьшится.

Глава 3

Простая сортировка

Когда в вашей базы данных накопится сколько-нибудь значительный объем данных, неизбежно возникнет необходимость в их сортировке. Имена потребуется отсортировать по алфавиту, студентов — по среднему баллу, клиентов — по регионам, данные о продажах — по ценам, города — по населению, страны — по ВВП и т. д.

Сортировка данных также может стать необходимым подготовительным шагом перед поиском. Как было показано в главе 2, «Массивы», двоичный поиск, применимый только к упорядоченным данным, выполняется намного быстрее линейного поиска.

Так как сортировка играет важную роль при работе с данными и при этом может быть весьма затратной по времени операцией, на эту тему велись обширные исследования в области информатики. Были разработаны некоторые весьма изощренные методы сортировки. В этой главе мы рассмотрим три простейших алгоритма: пузырьковую сортировку, сортировку методом выбора и сортировку методом вставки. Для демонстрации каждого метода создается отдельное приложение. В главе 7, «Нетривиальная сортировка», будут рассмотрены более совершенные алгоритмы: сортировка Шелла и быстрая сортировка.

Тривиальные методы, описанные в этой главе, работают относительно медленно и все же их стоит изучить. Они не только проще для понимания, но и лучше подходят в некоторых обстоятельствах. Например, сортировка методом вставки превосходит быструю сортировку для небольших и почти отсортированных файлов. Собственно, сортировка методом вставки часто используется как составная часть реализации быстрой сортировки.

В примерах программ этой главы используются классы массивов, разработанные в предыдущей главе. Алгоритмы сортировки реализуются в форме методов аналогичных классов.

Обязательно опробуйте приложения, представленные в этой главе. Они гораздо эффективнее объясняют, как работают алгоритмы сортировки, чем любые текстовые описания и статические картинки.

Как это делается?

Представьте, что ваша бейсбольная команда (см. главу 1, «Общие сведения») выстроилась на поле, как показано на рис. 1.3. На тренировку явились 9 игроков из основного состава и один запасной. Вы хотите построить игроков по росту (чтобы самый низкий игрок стоял слева). Как это сделать?



Рис. 3.1. Бейсбольная команда

Человеку проще, чем компьютерной программе — вы видите всех игроков сразу и можете немедленно отобрать самого высокого. Вам не нужно тщательно измерять и сравнивать всех по очереди. Кроме того, игроки не обязаны занимать строго определенные позиции. Они могут толкаться, пытаясь освободить побольше места, становиться друг за другом. После нескольких перестановок вы без труда выстроите всех игроков по росту (рис. 3.2).



Рис. 3.2. Бейсбольная команда, упорядоченная по росту

Программа в отличие от нас не «видит» данные. Она может сравнивать только двух игроков, потому что операторы сравнения работают именно по такому принципу. «Близорукость» некоторых алгоритмов постоянно приходится учитывать при анализе. Нам, людям, эта задача кажется простой, но алгоритм не видит «общей картины», а следовательно, он должен сосредоточиться на подробностях и следовать некоторым простым правилам.

Все три алгоритма, представленные в этой главе, состоят из двух шагов, которые повторяются снова и снова, пока данные не будут отсортированы:

1. Сравнить два элемента.
2. Поменять элементы местами или скопировать один из них.

Тем не менее разные алгоритмы по-разному выполняют эти операции.

Пузырьковая сортировка

Пузырьковая сортировка известна своей низкой скоростью, однако на концептуальном уровне — это простейший алгоритм сортировки, и поэтому мы начнем изучение алгоритмов сортировки именно с него.

Пример пузырьковой сортировки

Представьте, что вы близоруки (как компьютерная программа) и видите не более двух игроков одновременно и только если они стоят вплотную друг к другу. Как бы вы отсортировали игроков при таких ограничениях? Допустим, команда состоит из N игроков, а их позиции пронумерованы от 0 до N–1.

Алгоритм пузырьковой сортировки работает так: вы подходите к левому краю шеренги (позиция 0) и сравниваете двух игроков в позициях 0 и 1. Если левый игрок (позиция 0) выше, вы меняете их местами. Если выше правый игрок, они остаются на своих местах. Затем вы переходите на одну позицию вправо и сравниваете игроков в позициях 1 и 2. И снова, если левый игрок выше, вы меняете их местами. Схема сортировки показана на рис. 3.3.

Сортировка выполняется по следующим правилам:

1. Сравнить двух игроков.
2. Если левый игрок выше, поменять их местами.
3. Перейти на одну позицию вправо.

Перестановки продолжаются до тех пор, пока не будет достигнут правый край шеренги. Сортировка еще не завершена, но по крайней мере самый высокий игрок стоит в крайней правой позиции. Это происходит всегда: как только вы встретите самого высокого игрока, вы будете переставлять его при каждом сравнении, пока он (в конечном итоге) не перейдет в крайнюю правую позицию. Именно поэтому алгоритм называется «пузырьковой сортировкой»: в ходе его работы самый большой элемент, словно пузырек в жидкости, всплывает до конца массива. На рис. 3.4 показано, как выглядит строй после завершения первого прохода.

При первом проходе по всем данным выполняется N–1 сравнений и от 0 до N–1 перестановок (в зависимости от начального расположения игроков). Элемент в конце массива находится на своем месте и снова перемещаться уже не будет.

Теперь алгоритм возвращается к началу шеренги и начинает следующий проход. Он снова последовательно перемещается слева направо, сравнивая элементы и переставляя их в случае необходимости. На этот раз перебор останавливается за один элемент до конца, в позиции N–2, потому что последняя позиция (N–1) уже гарантированно содержит наибольший элемент.

Это правило можно сформулировать следующим образом:

4. После того как первый отсортированный элемент окажется на своем месте, вернуться к началу массива.

Процесс продолжается до тех пор, пока все элементы не будут упорядочены. Описать этот процесс намного сложнее, чем продемонстрировать его в действии, так что давайте просто понаблюдаем за работой приложения BubbleSort Workshop.

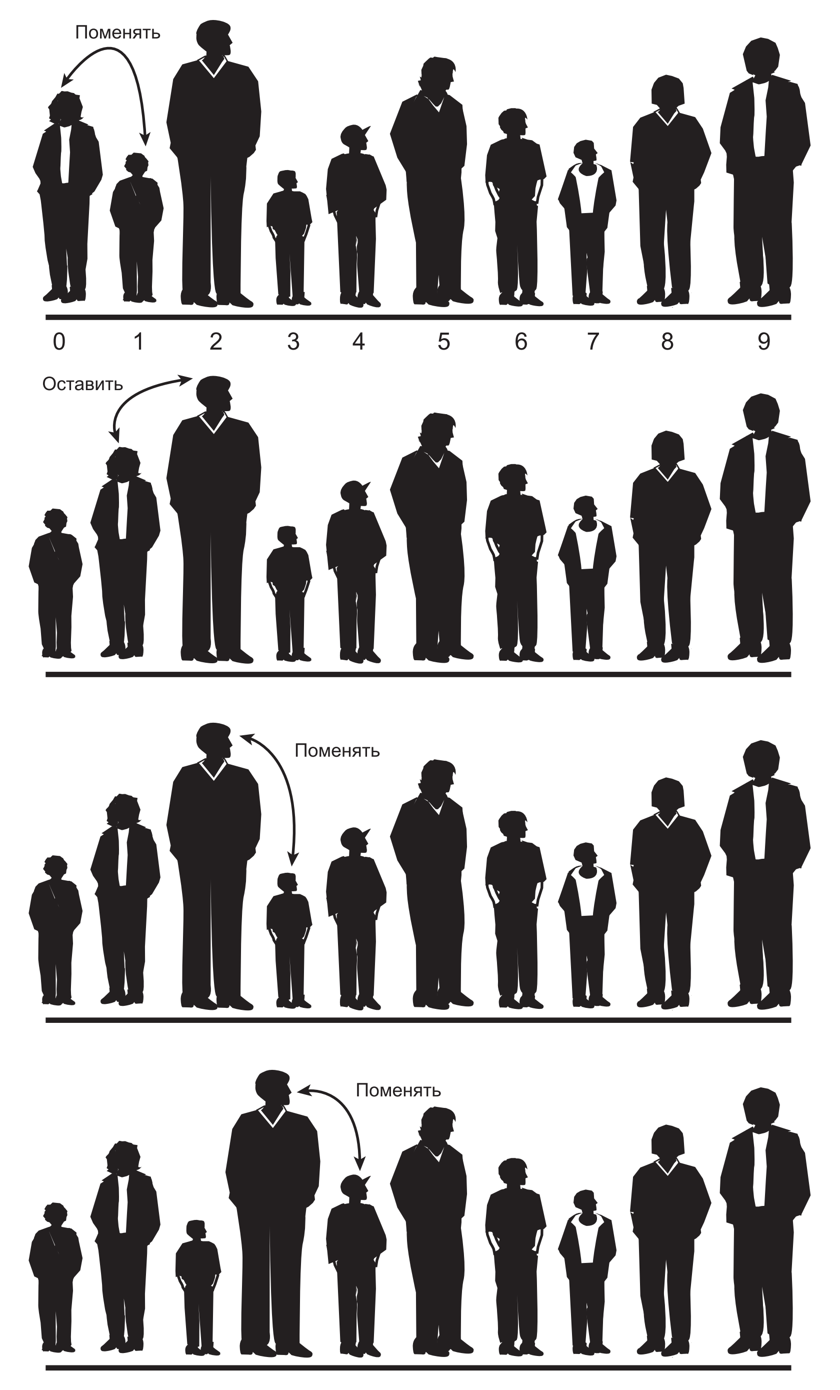


Рис. 3.3. Пузырьковая сортировка: начало первого прохода



Рис. 3.4. Пузырьковая сортировка: конец первого прохода

Приложение BubbleSort Workshop

Запустите приложение BubbleSort Workshop. В нем отображается некое подобие гистограммы со случайной высотой столбцов (рис. 3.5).

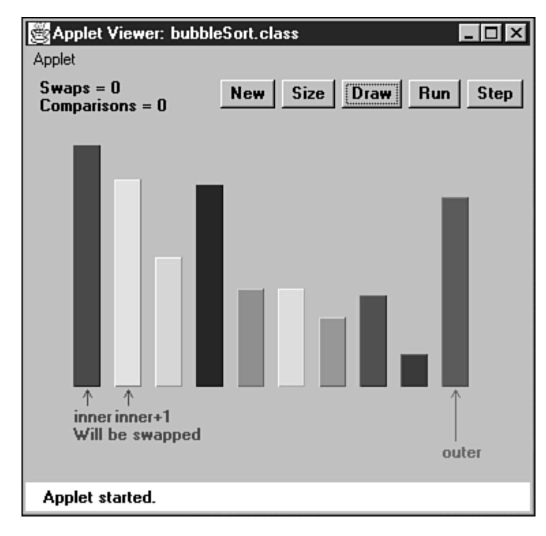


Рис. 3.5. Приложение BubbleSort Workshop

Кнопка Run

В приложении предусмотрены два режима выполнения: пользователь либо запускает сортировку в автоматическом режиме, либо проводит ее в пошаговом режиме. Чтобы получить общее представление о происходящем, щелкните на кнопке Run. Алгоритм выполняет пузырьковую сортировку столбцов по высоте. После завершения работы (секунд за 10 или около того) столбцы размещаются в упорядоченном состоянии, как показано на рис. 3.6.

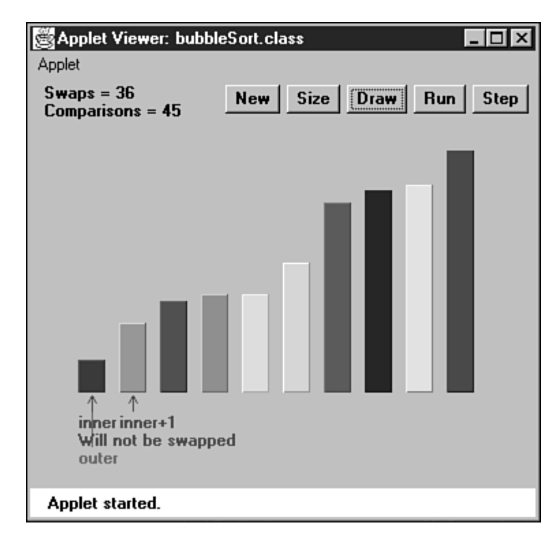


Рис. 3.6. После пузырьковой сортировки

Кнопка New

Чтобы провести сортировку заново, щелкните на кнопке New. Приложение создает новый набор столбцов и инициализирует алгоритм сортировки. Повторные нажатия New осуществляют переключения между двумя вариантами гистограммы: случайным, показанным на рис. 3.5, и изначальной сортировкой столбцов в обратном порядке. Последний вариант обычно обеспечивает максимальное количество операций во многих алгоритмах сортировки.

Кнопка Step

Польза от приложения BubbleSort Workshop в полной мере проявляется при пошаговой сортировке. Вы можете во всех подробностях понаблюдать за тем, как алгоритм выполняет каждый шаг.

Начните с создания новой случайной гистограммы кнопкой New. В окне приложения отображаются три стрелки, указывающие на разные столбцы. Две стрелки (inner и inner+1) находятся вблизи друг от друга слева. Еще одна стрелка — outer — расположена у правого края. (Подписи соответствуют именам счетчиков inner и outer во вложенных циклах, используемых в работе алгоритма.)

Щелкните на кнопке Step. Стрелки inner и inner+1 смещаются на одну позицию вправо, а столбцы могут поменяться местами. В примере с бейсбольной командой стрелки соответствуют двум игрокам, которых вы сравниваете и меняете местами в зависимости от результата проверки.

Текст под стрелками сообщает, будут ли переставляться элементы inner и inner+1, но это и так легко определить по относительной высоте столбцов: если слева находится более высокий столбец, то они поменяются местами. В верхней части

гистограммы выводится количество перестановок и сравнений, выполненных до настоящего момента. (Полная сортировка 10 столбцов требует 45 сравнений и в среднем 22 перестановок).

Продолжайте нажимать кнопку Step. Каждый раз, когда стрелки inner и inner+1 доходят от позиции 0 до outer, указатель outer перемещается на одну позицию влево. В процессе сортировки все столбцы справа от outer находятся на своих окончательных позициях, а столбцы слева (и в позиции outer) еще находятся в процессе сортировки.

Кнопка Size

Кнопка Size переключает размер массива (10 или 100 столбцов). На рис. 3.7 показано, как выглядит гистограмма из 100 случайных столбцов.

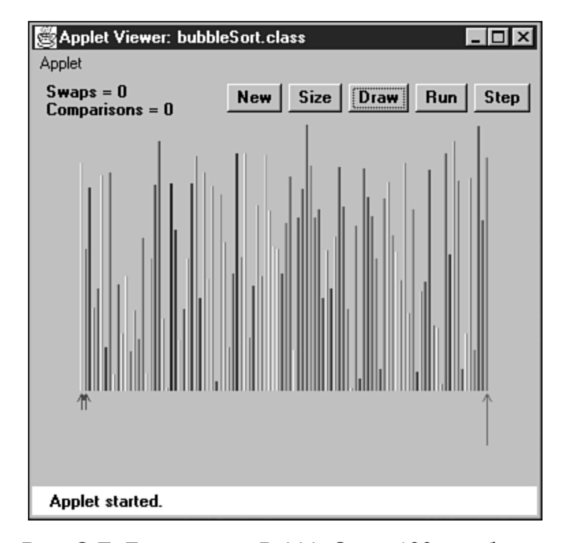


Рис. 3.7. Приложение Bubble Sort с 100 столбцами

Скорее всего, вам не захочется вручную управлять сортировкой 100 столбцов (разве что если вы отличаетесь выдающимся терпением). Нажмите кнопку Run и проследите за тем, как синие стрелки inner и inner+1 находят самый высокий неотсортированный столбец и перемещают его вправо, размещая слева от ранее отсортированных столбцов.

На рис. 3.8 показана незавершенная сортировка. Столбцы справа от красной (самой длинной) стрелки отсортированы. Столбцы слева от нее выглядят не так хаотично, но сортировка еще не завершена.

Если сортировка была запущена кнопкой Run и вы не можете уследить за летающими туда-сюда столбцами, процесс можно в любой момент остановить кнопкой Step. Далее сортировка либо продолжается в пошаговом режиме, либо повторное нажатие Run снова включает автоматический режим.

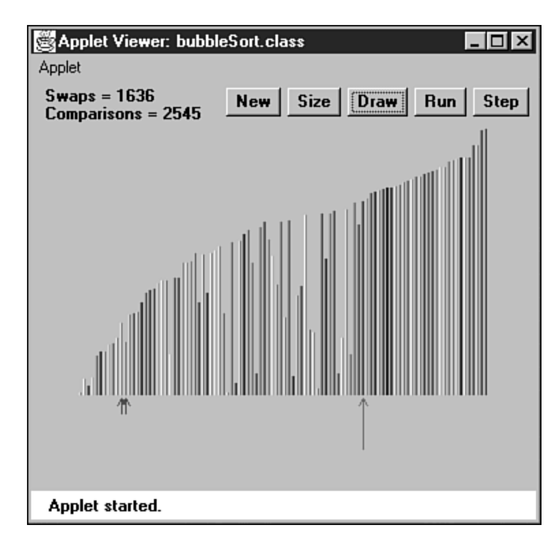


Рис. 3.8. 100 частично отсортированных столбцов

Кнопка Draw

Иногда во время сортировки в автоматическом режиме компьютер тратит время на выполнение других операций, и некоторые столбцы могут оказаться непрорисованными. Если это произойдет, нажмите кнопку Draw, чтобы выполнить принудительную перерисовку всех столбцов. Сортировка при этом приостанавливается, и для ее продолжения необходимо снова нажать кнопку Run.

Реализация пузырьковой сортировки на языке Java

В программе bubbleSort.java, приведенной в листинге 3.1, класс ArrayBub инкапсулирует массив a[], хранящий переменные типа long. В реальной программе данные, скорее всего, будут состоять из объектов, но мы используем примитивный тип для простоты. (Сортировка объектов продемонстрирована в программе objectSort.java в листинге 3.4.) Кроме того, для сокращения объема кода в листинг не включены методы find() и delete() класса ArrayBub, хотя обычно они присутствуют в подобных классах.

Листинг 3.1. Программа bubbleSort.java

```
// bubbleSort.java
// Пузырьковая сортировка
// Запуск программы: C>java BubbleSortApp
////////////////////////////////////////////////////////////////
class ArrayBub
   {
   private long[] a;                 // Ссылка на массив a
```

```
   private int nElems;               // Количество элементов данных
//--------------------------------------------------------------
   public ArrayBub(int max)          // Конструктор
      {
      a = new long[max];             // Создание массива
      nElems = 0;                    // Пока нет ни одного элемента
      }
//--------------------------------------------------------------
   public void insert(long value)    // Вставка элемента в массив
      {
      a[nElems] = value;             // Собственно вставка
      nElems++;                      // Увеличение размера
      }
//--------------------------------------------------------------
   public void display()             // Вывод содержимого массива
      {
      for(int j=0; j<nElems; j++)       // Для каждого элемента
         System.out.print(a[j] + " ");  // Вывод
      System.out.println("");
      }
//--------------------------------------------------------------
   public void bubbleSort()
      {
      int out, in;
      for(out=nElems-1; out>1; out--)   // Внешний цикл (обратный)
         for(in=0; in<out; in++)        // Внутренний цикл (прямой)
            if( a[in] > a[in+1] )       // Порядок нарушен?
               swap(in, in+1);          // Поменять местами
      }
//--------------------------------------------------------------
   private void swap(int one, int two)
      {
      long temp = a[one];
      a[one] = a[two];
      a[two] = temp;
      }
//--------------------------------------------------------------
   }  // Конец класса ArrayBub
////////////////////////////////////////////////////////////////
class BubbleSortApp
   {
   public static void main(String[] args)
      {
      int maxSize = 100;            // Размер массива
      ArrayBub arr;                 // Ссылка на массив
      arr = new ArrayBub(maxSize);  // Создание массива

      arr.insert(77);               // Вставка 10 элементов
      arr.insert(99);
```

продолжение ➪

Листинг 3.1 (*продолжение*)

```
      arr.insert(44);
      arr.insert(55);
      arr.insert(22);



      arr.insert(88);
      arr.insert(11);
      arr.insert(00);
      arr.insert(66);
      arr.insert(33);

      arr.display();               // Вывод элементов

      arr.bubbleSort();            // Пузырьковая сортировка элементов
      arr.display();               // Повторный вывод
      }  //
   }  // Конец класса BubbleSortApp
////////////////////////////////////////////////////////////////
```

Конструктор и методы insert() и display() этого класса сходны с теми, которые приводились ранее. Однако в классе появился новый метод bubbleSort(). При вызове этого метода содержимое массива переставляется в отсортированном порядке.

Метод main() вставляет в массив 10 элементов в случайном порядке, выводит содержимое массива, сортирует его вызовом bubbleSort(), а затем выводит снова. Результат выполнения:

```
77 99 44 55 22 88 11 0 66 33
0 11 22 33 44 55 66 77 88 99
```

Основной код метода bubbleSort() состоит всего из четырех строк:

```
public void bubbleSort()
   {
   int out, in;
   for(out=nElems-1; out>1; out--)   // Внешний цикл (обратный)
      for(in=0; in<out; in++)        // Внутренний цикл (прямой)
         if( a[in] > a[in+1] )       // Порядок нарушен?
            swap(in, in+1);          // Поменять местами
   }
```

В результате сортировки наименьший элемент должен находиться в начале массива (индекс 0), а наибольший — в конце (индекс nElems-1). Счетчик цикла out внешнего цикла for начинает перебор с конца массива (в позиции nElems-1) и последовательно уменьшается в процессе перебора. Элементы с индексами, большими out, находятся на своих окончательных местах. Переменная out смещается влево после каждого прохода, чтобы алгоритм не затрагивал уже отсортированные элементы.

Счетчик in начинается с начала массива и последовательно увеличивается при каждой итерации внутреннего цикла вплоть до out. Внутренний цикл сравнивает две ячейки, на которые указывают in и in+1, и если элемент в ячейке in больше элемента в ячейке in+1 — меняет их местами.

Для наглядности перестановка элементов выполняется отдельным методом `swap()`. Метод просто меняет местами значения в двух ячейках массива; значение первой ячейки сохраняется во временной переменной, на его место записывается значение из второй ячейки, после чего содержимое временной переменной копируется во вторую ячейку. Вообще говоря, использовать отдельный метод `swap()` в реальной программе нежелательно, потому что вызов функции сопряжен с дополнительными, пусть и небольшими затратами времени. Если вы пишете собственную реализацию сортировки, лучше выполните перестановку «на месте».

Инварианты

Во многих алгоритмах задействованы условия, которые остаются неизменными в ходе выполнения алгоритма. Такие условия называются *инвариантами*. Умение распознавать инварианты поможет понять суть алгоритма. Инварианты также могут пригодиться в ходе отладки; вы можете периодически проверять истинность инвариантов и выдавать сообщение об ошибке в случае их нарушения.

В программе `bubbleSort.java` инвариант определяет, что элементы данных справа от `out` отсортированы. Это условие остается истинным. (При первом проходе отсортированные элементы отсутствуют, а справа от `out` нет ни одного элемента, так как счетчик инициализируется позицией крайнего правого элемента.)

Сложность пузырьковой сортировки

Итак, в приложении BubbleSort Workshop с 10 столбцами на первом проходе выполняется девять сравнений, на втором — восемь и так далее, вплоть до одного сравнения на последнем проходе. Для 10 элементов в сумме получается:

$$9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 45.$$

В общем виде, если массив состоит из N элементов, на первом проходе выполняются $N-1$ сравнений, на втором — $N-2$ и т. д. Сумма такого ряда вычисляется по формуле:

$$(N-1) + (N-2) + (N-3) + ... + 1 = N \times (N-1)/2.$$

Если $N = 10$, то сумма равна 45 ($10 \times 9/2$).

Таким образом, алгоритм выполняет около $N^2/2$ сравнений (на -1 можно не обращать внимания, особенно при больших N).

Количество перестановок меньше количества сравнений, потому что два столбца переставляются только если того требует алгоритм. Для случайных данных перестановка в среднем выполняется в половине случаев, поэтому количество перестановок составит около $N^2/4$. (Хотя в худшем случае, если данные изначально отсортированы в обратном порядке, перестановка будет выполняться при каждом сравнении).

Количество как перестановок, так и сравнений пропорционально N^2. Так как константы в O-синтаксисе не учитываются, можно сказать, что пузырьковая сортировка выполняется за время $O(N^2)$. Это достаточно медленно — в этом нетрудно убедиться, запустив приложение BubbleSort Workshop для 100 столбцов.

Каждый раз, когда вы видите вложенные циклы (как в алгоритме пузырьковой сортировки или в другом алгоритме сортировки этой главы), можно предположить, что алгоритм выполняется за время $O(N^2)$. Внешний цикл выполняется N раз, внутренний цикл — тоже N раз для каждой итерации внешнего цикла (или N, разделенное на константу). Таким образом, всего будет выполнено $N \times N$, или N^2 итераций.

Сортировка методом выбора

Алгоритм сортировки методом выбора превосходит пузырьковую сортировку по характеристикам — количество необходимых перестановок сокращается с $O(N^2)$ до $O(N)$. К сожалению, количество сравнений остается равным $O(N^2)$. Все же сортировка методом выбора обладает значительными преимуществами для больших записей, которые необходимо физически перемещать в памяти, в результате чего время перестановки оказывается намного более важным, чем время сравнения. (Это не относится к языку Java, в котором перемещаются ссылки, а не целые объекты.)

Пример сортировки методом выбора

Вернемся к примеру с бейсбольной командой. При сортировке методом выбора сравнение уже не ограничивается игроками, стоящими рядом друг с другом. Следовательно, вам понадобится как-то сохранить рост игрока; например, можно записать его в блокноте.

Краткое описание

В сортировке методом выбора вы последовательно перебираете всех игроков и выбираете (отсюда и название) самого низкорослого из них. Этот игрок меняется местами с тем, который стоит в крайней левой позиции (0). Левый игрок отсортирован, и в дальнейшем уже перемещаться не будет. Обратите внимание: в этом алгоритме отсортированные игроки собираются слева (нижние индексы), тогда как при пузырьковой сортировке они собираются справа.

Следующий проход начинается с позиции 1, а обнаруженный минимальный элемент меняется местами с элементом в позиции 1. Процесс продолжается до тех пор, пока не будут отсортированы все игроки.

Более подробное описание

Перебор начинается от левого края шеренги игроков. Запишите рост крайнего левого игрока в блокнот и положите на землю полотенце перед этим игроком. Затем сравните рост следующего игрока с ростом, записанным в блокноте. Если новый игрок меньше, вычеркните рост первого игрока и впишите вместо него рост второго игрока — и переложите полотенце перед новым «самым низкорослым» (на данный момент) игроком. Продолжайте двигаться вдоль шеренги, сравнивая рост игроков с записанным минимумом. Обнаружив более низкого игрока, изменяйте записанное в блокноте значение и перекладывайте полотенце. Когда проход будет закончен, полотенце будет лежать перед самым низкорослым игроком.

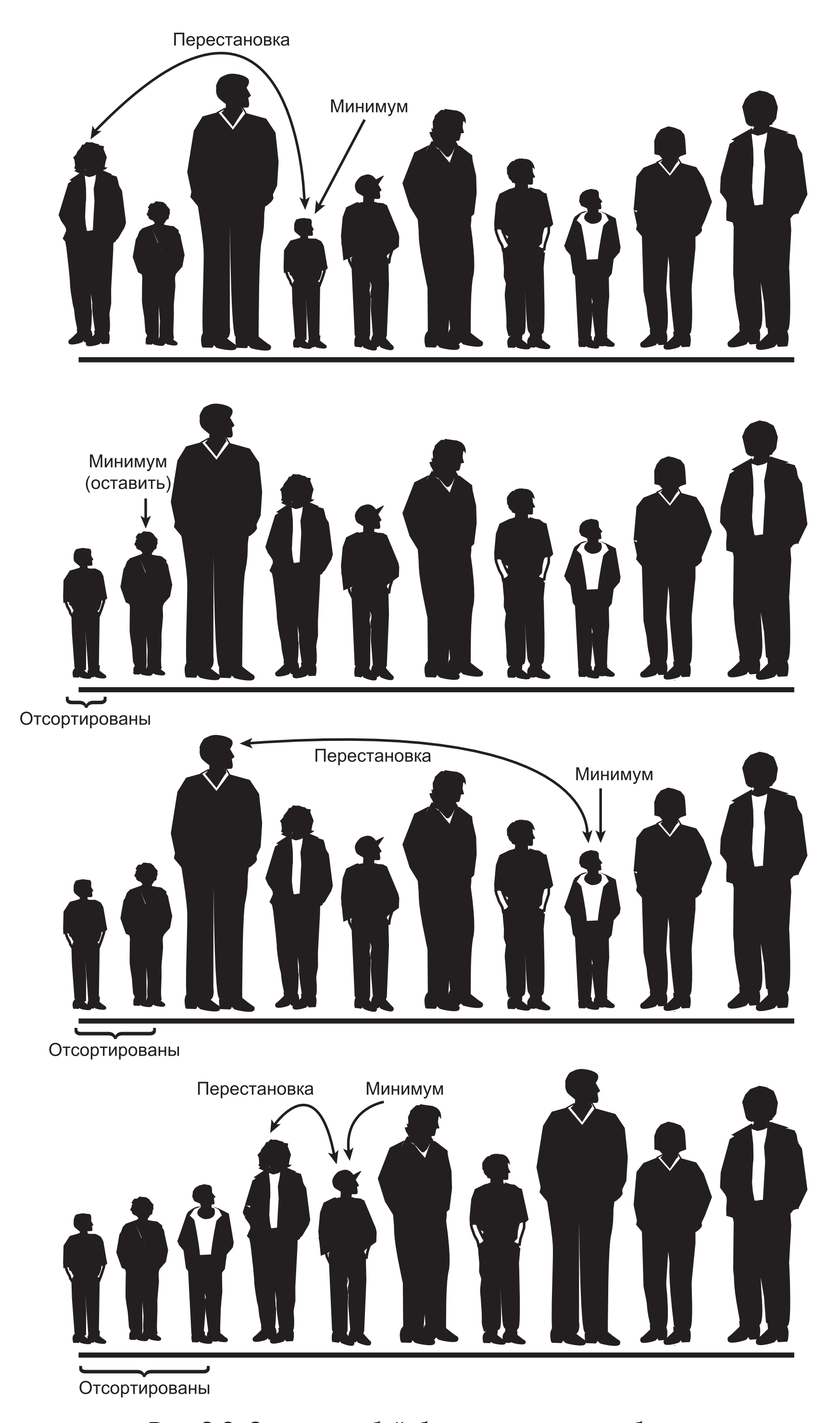


Рис. 3.9. Сортировка бейсболистов методом выбора

Переставьте этого игрока с игроком, находящимся с левого края шеренги. Один игрок отсортирован. Вы провели $N-1$ сравнений, но только одну перестановку. С каждым последующим проходом еще один игрок сортируется и переходит на левый край, и поиск следующего минимума производится среди меньшего количества игроков. На рис. 3.9 показано, как выглядит результат сортировки после первых трех проходов.

Приложение SelectSort Workshop

Чтобы увидеть, как работает сортировка методом выбора, запустите приложение SelectSort Workshop. Кнопки выполняют те же функции, что и в приложении BubbleSort. Кнопка `New` создает новый массив из 10 случайно расположенных столбцов. Красная стрелка с подписью `outer` изначально находится слева; она указывает на крайний левый неупорядоченный столбец. Постепенно она будет смещаться вправо по мере добавления в отсортированную подгруппу слева новых столбцов.

Малиновая стрелка `min` также изначально указывает на крайний левый столбец; она перемещается, отмечая самый низкий столбец, обнаруженный до настоящего момента. (Малиновая стрелка `min` соответствует полотенцу в нашей бейсбольной аналогии.) Синей стрелкой `inner` отмечается столбец, который в данный момент сравнивается с минимумом.

При повторных нажатиях кнопки `Step` стрелка `inner` перемещается слева направо, последовательно проверяя каждый столбец и сравнивая его со столбцом, на который указывает `min`. Если столбец `inner` короче, стрелка `min` перемещается к новому, более короткому столбцу. Когда стрелка `inner` достигает правого края, `min` указывает на самый короткий из всех неупорядоченных столбцов. Этот столбец меняется местами с `outer`, крайним левым из неупорядоченных столбцов.

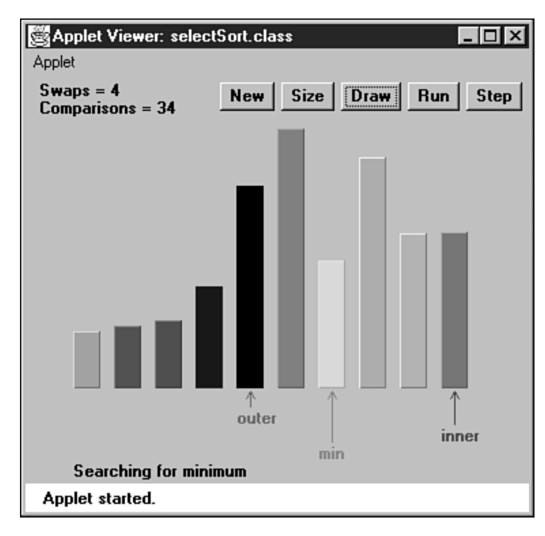


Рис. 3.10. Приложение SelectSort Workshop

На рис. 3.10 показано состояние сортировки на середине процесса. Столбцы слева от outer отсортированы, а стрелка inner добралась до крайней правой позиции в поисках самого короткого столбца. Стрелка min зарегистрировала позицию этого столбца, который поменяется местами с outer.

Используйте кнопку Size, чтобы переключиться на гистограмму с 100 столбцами, и отсортируйте новый случайный набор. Вы увидите, как стрелка min некоторое время остается у потенциального минимума, а потом переходит к новому столбцу, когда синяя стрелка inner обнаружит меньшего кандидата. Красная стрелка outer медленно, но непоколебимо движется вправо по мере накопления отсортированных столбцов слева.

Реализация сортировки методом выбора на языке Java

Листинг программы selectSort.java похож на листинг bubbleSort.java, разве что контейнерный класс называется ArraySel вместо ArrayBub, а метод bubbleSort() заменен методом selectSort(). Вот как он выглядит:

```
public void selectionSort()
   {
   int out, in, min;

   for(out=0; out<nElems-1; out++)   // Внешний цикл
      {
      min = out;                     // Минимум
      for(in=out+1; in<nElems; in++) // Внутренний цикл
         if(a[in] < a[min] )         // Если значение min больше,
            min = in;                // значит, найден новый минимум
      swap(out, min);                // Поменять их местами
      }
   }
```

Внешний цикл (переменная out) начинает перебор с начала массива (индекс 0) и продвигается к большим значениям индексов. Внутренний цикл (переменная in) начинается с out и также двигается вправо.

В каждой новой позиции in сравниваются элементы a[in] и a[min]. Если a[in] меньше, то min присваивается значение in. В конце внутреннего цикла min указывает на минимальное значение текущего прохода, и алгоритм переставляет элементы массива, на которые ссылаются out и min.

В листинге 3.2 приведен полный код программы selectSort.java.

Листинг 3.2. Программа selectSort.java

```
// selectSort.java
// Сортировка методом выбора
// Запуск программы: C>java SelectSortApp
////////////////////////////////////////////////////////////////
```

продолжение ⇨

Листинг 3.2 (*продолжение*)

```
class ArraySel
   {
   private long[] a;                 // Ссылка на массив a
   private int nElems;               // Количество элементов данных
//--------------------------------------------------------------
   public ArraySel(int max)          // Конструктор
      {
      a = new long[max];             // Создание массива
      nElems = 0;                    // Пока нет ни одного элемента
      }
//--------------------------------------------------------------
   public void insert(long value)    // Вставка элемента в массив
      {
      a[nElems] = value;             // Собственно вставка
      nElems++;                      // Увеличение размера
      }
//--------------------------------------------------------------
   public void display()             // Вывод содержимого массива
      {
      for(int j=0; j<nElems; j++)       // Для каждого элемента
         System.out.print(a[j] + " ");  // Вывод
      System.out.println("");
   }
//--------------------------------------------------------------
   public void selectionSort()
      {
      int out, in, min;
      for(out=0; out<nElems-1; out++)   // Внешний цикл
         {
         min = out;                     // Минимум
         for(in=out+1; in<nElems; in++) // Внутренний цикл
            if(a[in] < a[min] )         // Если значение min больше,
               min = in;                // значит, найден новый минимум
         swap(out, min);                // swap them
         }
      }
//--------------------------------------------------------------
   private void swap(int one, int two)
      {
      long temp = a[one];
      a[one] = a[two];
      a[two] = temp;
   }
//--------------------------------------------------------------
   }
////////////////////////////////////////////////////////////////
```

```
class SelectSortApp
   {
   public static void main(String[] args)
      {
      int maxSize = 100;            // Размер массива
      ArraySel arr;                 // Ссылка на массив
      arr = new ArraySel(maxSize);  // Создание массива
      arr.insert(77);               // Вставка 10 элементов
      arr.insert(99);
      arr.insert(44);
      arr.insert(55);
      arr.insert(22);
      arr.insert(88);
      arr.insert(11);
      arr.insert(00);
      arr.insert(66);
      arr.insert(33);

      arr.display();                // Вывод элементов

      arr.selectionSort();          // Сортировка методом выбора

      arr.display();                // Повторный вывод
      }
   }  // Конец класса SelectSortApp
////////////////////////////////////////////////////////////////
```

Вывод `selectSort.java` идентичен выводу `bubbleSort.java`:

```
77 99 44 55 22 88 11 0 66 33
0 11 22 33 44 55 66 77 88 99
```

Инвариант

В программе `selectSort.java` элементы данных с индексами, меньшими либо равными `out`, всегда отсортированы.

Сложность сортировки методом выбора

Сортировка методом выбора выполняет такое же количество сравнений, что и пузырьковая сортировка: $N \times (N-1)/2$. Для каждых 10 элементов данных выполняются 45 сравнений, однако количество перестановок уменьшается. Для больших значений N время сравнения является определяющим фактором, поэтому мы можем сказать, что сортировка методом выбора выполняется за время $O(N^2)$, как и пузырьковая сортировка. Тем не менее совершенно очевидно, что сортировка методом выбора выполняется быстрее из-за меньшего количества перестановок. Для меньших значений N сортировка методом выбора может работать заметно быстрее, особенно если перестановка выполняется намного медленнее сравнения.

Сортировка методом вставки

В большинстве случаев сортировка методом вставки является лучшим из элементарных алгоритмов сортировки, описанных в этой главе. Она также выполняется за время $O(N^2)$, но работает примерно вдвое быстрее пузырьковой сортировки, а в обычных ситуациях немного быстрее сортировки методом выбора. Кроме того, сортировка методом вставки не слишком сложна, хотя ее алгоритм немного сложнее двух других. Она часто используется на завершающей стадии более сложных алгоритмов, например быстрой сортировки.

Пример сортировки методом вставки

В исходном состоянии игроки бейсбольной команды выстроены в случайном порядке. Сортировку методом вставки легче описывать с середины процесса, когда команда уже наполовину отсортирована.

Частичная сортировка

На этой стадии где-то в середине шеренги устанавливается воображаемый маркер. (Допустим, вы положили красную футболку на землю перед игроком.) Игроки слева от маркера *частично отсортированы*; это означает, что каждый из них выше своего соседа слева. Однако это вовсе не означает, что игроки занимают свои итоговые позиции, потому что они, возможно, будут перемещены в результате вставки других, еще не отсортированных игроков.

Обратите внимание: ситуация частичной сортировки не возникает в ходе пузырьковой сортировки и сортировки методом выбора. В этих алгоритмах подгруппа элементов данных полностью упорядочена в любой конкретный момент времени; в сортировке методом вставки подгруппа элементов упорядочена только частично.

Помеченный игрок

Игрок, рядом с которым лежит маркер (будем называть его «помеченным» игроком), и все игроки справа от него еще не отсортированы (рис. 3.11, *а*).

Здесь мы собираемся вставить помеченного игрока в подходящее место (частично) отсортированной подгруппы. Однако для этого необходимо сдвинуть некоторых игроков подгруппы вправо, чтобы освободить место для вставки. Для этого помеченный игрок выводится из шеренги (в программе соответствующий элемент данных сохраняется во временной переменной) — этот шаг показан на рис. 3.11, *б*.

Теперь отсортированные игроки сдвигаются, чтобы освободить место в частично отсортированной подгруппе. Самый высокий игрок переходит на место помеченного игрока, следующий по росту — на место самого высокого и т. д.

Когда следует остановить перемещение? Представьте, что вы и помеченный игрок двигаетесь влево по шеренге. В каждой позиции игрок перемещается вправо, но вы также сравниваете помеченного игрока с текущим. Процесс перемещения останавливается тогда, когда будет перемещен последний игрок, рост которого

превышает рост помеченного игрока. Последнее перемещение открывает свободное место для помеченного игрока, который после вставки окажется в правильном порядке частичной сортировки. Этот шаг показан на рис. 3.11, *в*.

Размер частично отсортированной подгруппы увеличивается на одного игрока, а неупорядоченная подгруппа становится на одного игрока меньше. Маркер (футболка) перемещается на одну позицию вправо, и снова оказывается перед крайним левым игроком в неупорядоченной группе. Процесс повторяется до тех пор, пока все неупорядоченные игроки не будут вставлены (отсюда и название алгоритма) в соответствующие позиции частично отсортированной подгруппы.

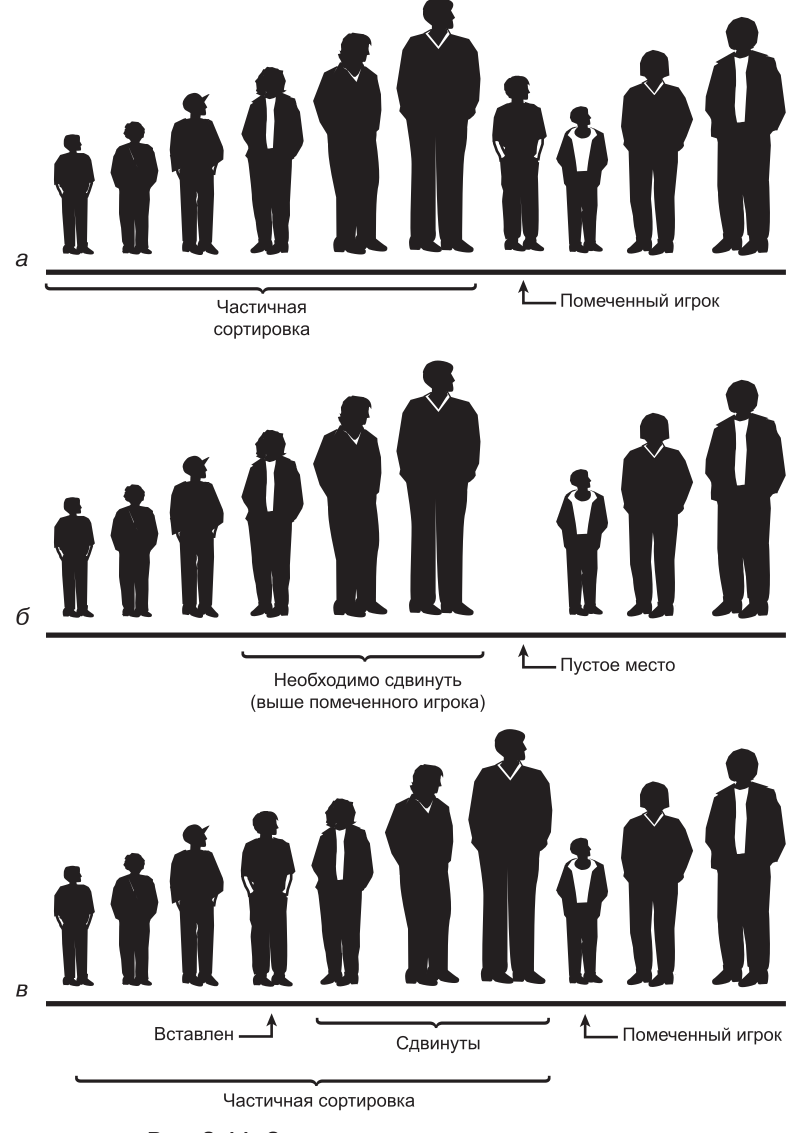


Рис. 3.11. Сортировка игроков методом вставки

Приложение InsertSort Workshop

Приложение InsertSort Workshop демонстрирует сортировку методом вставки. В отличие от других приложений, вероятно, лучше начать со 100 случайных столбцов вместо 10 — результат будет более поучительным.

Сортировка 100 столбцов

Переключитесь на гистограмму из 100 столбцов (кнопка Size) и щелкните на кнопке Run, чтобы понаблюдать за сортировкой столбцов. Вы увидите, что короткая красная стрелка outer отмечает границу между частично отсортированными столбцами слева и несортированными столбцами справа. Синяя стрелка inner двигается влево от позиции outer, пытаясь найти подходящее место для вставки помеченного столбца. На рис. 3.12 показано, как выглядит этот процесс, когда примерно половина столбцов частично отсортирована.

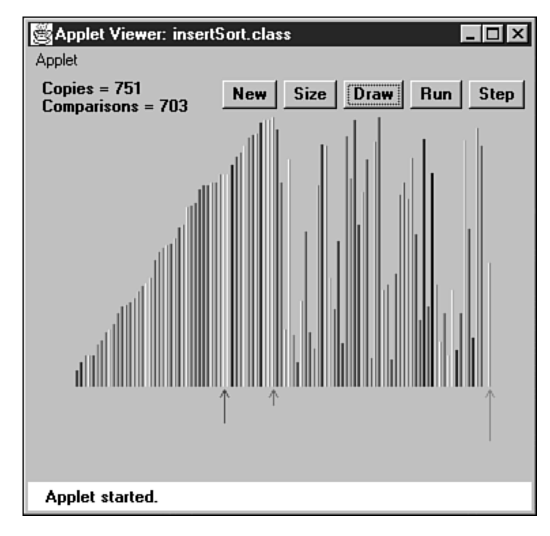


Рис. 3.12. Приложение InsertSort Workshop с 100 столбцами

Помеченный столбец хранится во временной переменной, на которую указывает малиновая стрелка у правого края гистограммы, но содержимое этой переменной изменяется так часто, что его трудно разглядеть (без переключения в пошаговый режим).

Сортировка 10 столбцов

Чтобы рассмотреть происходящее во всех подробностях, переключитесь в 10-столбцовый режим при помощи кнопки Size. (При необходимости воспользуйтесь кнопкой New, чтобы столбцы располагались в случайном порядке.)

В самом начале выполнения алгоритма inner и outer указывают на второй столбец слева (индекс 1), а в первом сообщении говорится о копировании outer в temp.

Копирование освобождает место для перемещения. (Стрелки inner-1 в приложении нет, но разумеется, она всегда должна находиться на одну позицию левее inner.)

Щелкните на кнопке Step. Элемент outer копируется в temp. При выполнении копирования приложение удаляет элемент из источника, оставляя его пустым. Это не совсем точно — в реальной Java-программе ссылка остается в источнике, однако очистка источника помогает лучше понять, что происходит в программе.

Дальнейшие действия зависят от того, расположены ли первые два столбца в порядке сортировки (меньший слева). Если это так, то выводится сообщение о том, что копирование не требуется. Если же первые два столбца находятся в обратном порядке, выводится сообщение о копировании inner-1 в inner. Это перемещение необходимо для того, чтобы освободить место для повторной вставки значения из temp. На первом проходе такое перемещение только одно; на последующих проходах потребуется больше перемещений. Ситуация показана на рис. 3.13.

При следующем щелчке inner-1 копируется в inner, а стрелка inner смещается на одну позицию влево. Процесс перемещения на этом завершается.

Какой бы из двух столбцов ни был короче, при следующем щелчке выводится сообщение о копировании temp в inner. Оно действительно выполняется, но если два столбца изначально находились в нужном порядке, последствия копирования незаметны — temp и inner содержат одни и те же данные. Казалось бы, копирование данных поверх тех же данных неэффективно, но алгоритм работает быстрее без проверки этой ситуации (которая встречается относительно редко).

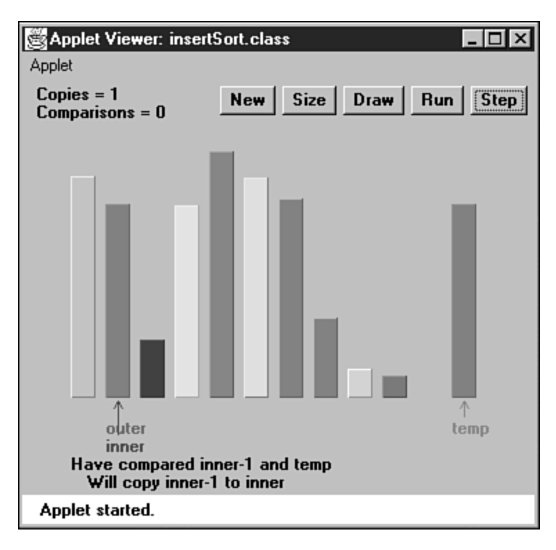


Рис. 3.13. Приложение InsertSort Workshop с 10 столбцами

Теперь первые два столбца частично отсортированы (то есть правильно упорядочены по отношению друг к другу), а стрелка outer перемещается на одну позицию вправо, к третьему столбцу (индекс 2). Процесс повторяется. На этом проходе возможны 0, 1 или 2 перемещения в зависимости от того, какое место третий столбец должен занимать среди двух первых.

Продолжайте процесс сортировки в пошаговом режиме. Происходящее становится более понятным после того, как процесс отработает некоторое время, и слева появятся отсортированные столбцы. Вы заметите, как после выполнения достаточного количества перемещений освобождается место для возвращения столбца из временной ячейки temp на положенное место.

Реализация сортировки методом вставки на языке Java

Следующий метод из программы insertSort.java выполняет сортировку методом вставки:

```
public void insertionSort()
   {
   int in, out;

   for(out=1; out<nElems; out++)     // out - разделительный маркер
      {
      long temp = a[out];            // Скопировать помеченный элемент
      in = out;                      // Начать перемещения с out
      while(in>0 && a[in-1] >= temp) // Пока не найден меньший элемент
         {
         a[in] = a[in-1];            // Сдвинуть элемент вправо
         --in;                       // Перейти на одну позицию влево
         }
      a[in] = temp;                  // Вставить помеченный элемент
      }
   }
```

Во внешнем цикле for счетчик начинает с позиции 1 и двигается вправо. Он отмечает крайний левый неотсортированный элемент. Во внутреннем цикле while счетчик in начинает с позиции out и двигается влево — либо пока temp не станет меньше элемента массива, либо когда дальнейшее смещение станет невозможным. При каждом проходе по циклу while следующий отсортированный элемент сдвигается на одну позицию вправо.

На первый взгляд трудно рассмотреть связь между операциями алгоритма InsertSort Workshop и кодом, поэтому на рис. 3.14 изображена блок-схема метода insertionSort() с соответствующими сообщениями приложения InsertSort Workshop. В листинге 3.3 приведен полный код программы insertSort.java.

Листинг 3.3. Программа insertSort.java

```
// insertSort.java
// Сортировка методом вставки
// Запуск программы: C>java InsertSortApp
//--------------------------------------------------------------
class ArrayIns
   {
   private long[] a;                 // Ссылка на массив a
```

```
   private int nElems;               // Количество элементов данных
//--------------------------------------------------------------
   public ArrayIns(int max)          // Конструктор
      {
      a = new long[max];             // Создание массива
      nElems = 0;                    // Пока нет ни одного элемента
      }
//--------------------------------------------------------------
   public void insert(long value)    // Вставка элемента в массив
      {
      a[nElems] = value;             // Собственно вставка
      nElems++;                      // Увеличение размера
      }
//--------------------------------------------------------------
   public void display()             // Вывод содержимого массива
      {
      for(int j=0; j<nElems; j++)       // Для каждого элемента
         System.out.print(a[j] + " ");  // Вывод
      System.out.println("");
      }
//--------------------------------------------------------------
   public void insertionSort()
      {
      int in, out;

      for(out=1; out<nElems; out++)     // out - разделительный маркер
         {
         long temp = a[out];            // Скопировать помеченный элемент
         in = out;                      // Начать перемещения с out
         while(in>0 && a[in-1] >= temp) // Пока не найден меньший элемент
            {
            a[in] = a[in-1];            // Сдвинуть элемент вправо
            --in;                       // Перейти на одну позицию влево
            }
         a[in] = temp;                  // Вставить помеченный элемент
         }
      }
//--------------------------------------------------------------
}  // Конец класса ArrayIns
////////////////////////////////////////////////////////////////
class InsertSortApp
   {
   public static void main(String[] args)
      {
      int maxSize = 100;            // Размер массива
      ArrayIns arr;                 // Ссылка на массив
      arr = new ArrayIns(maxSize);  // Создание массива

      arr.insert(77);               // Вставка 10 элементов
```

продолжение ⇗

Листинг 3.3 (*продолжение*)

```
      arr.insert(99);
      arr.insert(44);
      arr.insert(55);
      arr.insert(22);
      arr.insert(88);
      arr.insert(11);
      arr.insert(00);
      arr.insert(66);
      arr.insert(33);

      arr.display();              // Вывод элементов

      arr.insertionSort();        // Сортировка методом вставки

      arr.display();              // Повторный вывод
      }
   }
////////////////////////////////////////////////////////////////
```

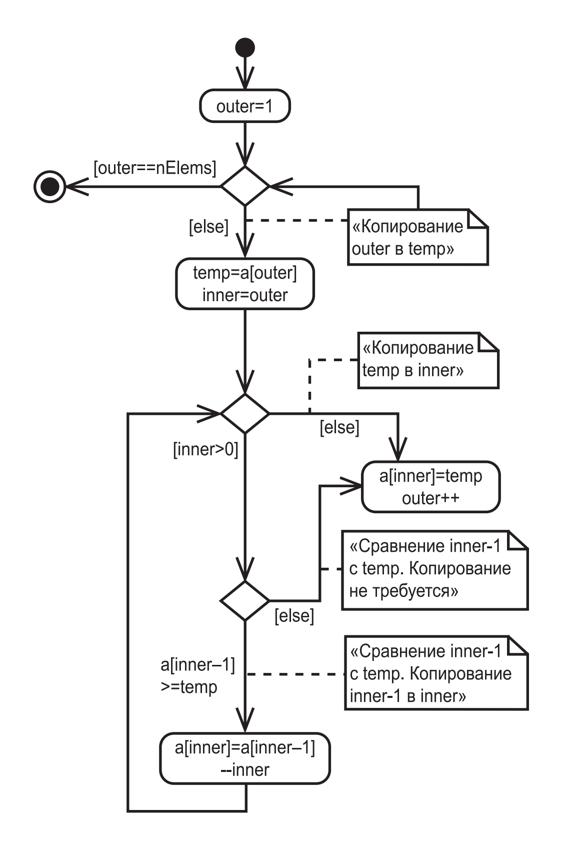


Рис. 3.14. Блок-схема метода insertSort()

Результат работы программы `insertSort.java` не отличается от результатов других программ этой главы:

```
77 99 44 55 22 88 11 0 66 33
0 11 22 33 44 55 66 77 88 99
```

Инварианты сортировки методом вставки

В конце каждого прохода, непосредственно после вставки элемента из `temp`, элементы данных с индексами, меньшими `outer`, являются частично отсортированными.

Сложность сортировки методом вставки

Сколько операций сравнения и копирования требует этот алгоритм? При первом проходе сравнивается не более одного элемента, на втором — не более двух и так далее, вплоть до $N-1$ сравнений на последнем проходе. В сумме получается:

$$1 + 2 + 3 + ... + N-1 = N \times (N-1)/2.$$

Тем не менее, поскольку при каждом проходе в среднем фактически сравнивается только половина от максимального количества элементов (прежде чем будет найдена точка вставки), сумму можно уменьшить вдвое; получается:

$$N \times N-1)/4.$$

Количество операций копирования приблизительно совпадает с количеством сравнений. Однако копирование занимает меньше времени, чем перестановка, так что для случайных данных этот алгоритм работает вдвое быстрее пузырьковой сортировки и быстрее сортировки методом выбора. Как бы то ни было, как и во всех остальных методах сортировки этой главы, для случайных данных сортировка методом вставки выполняется за время порядка $O(N^2)$.

Для данных, уже прошедших предварительную сортировку, этот алгоритм работает гораздо эффективнее. Если данные упорядочены, то условие цикла `while` никогда не бывает истинным; следовательно, оно вырождается в простую команду во внешним цикле, выполняемую $N-1$ раз. В этом случае алгоритм выполняется за время $O(N)$. Если данные почти отсортированы, то сортировка методом вставки выполняется почти за время $O(N)$, а следовательно, является простым и эффективным способом упорядочения файлов данных с небольшими отклонениями в порядке сортировки.

Но если данные изначально отсортированы в обратном порядке, алгоритму придется выполнить все возможные сравнения и перемещения, и тогда вставка методом сортировки выполняется не быстрее пузырьковой сортировки. Вы можете проверить этот факт, включив режим обратной сортировки данных (кнопка `New`) в приложении InsertSort Workshop.

Сортировка объектов

Для простоты все алгоритмы сортировки, рассмотренные до настоящего момента, применялись к примитивному типу данных long. Однако на практике сортировка чаще применяется к объектам, а не примитивным типам. Соответственно в листинге 3.4 приведена Java-программа objectSort.java, сортирующая массив объектов Person (см. программу classDataArray.java в главе 2).

Реализация сортировки объектов на языке Java

В следующей программе на языке Java используется алгоритм сортировки методом вставки из предыдущего раздела. Объекты Person сортируются по фамилии (lastName); это ключевое поле. Код программы objectSort.java приведен в листинге 3.4.

Листинг 3.4. Программа objectSort.java

```
// objectSort.java
// Сортировка объектов (с применением сортировки методом вставки)
// Запуск программы: C>java ObjectSortApp
////////////////////////////////////////////////////////////////
class Person
   {
   private String lastName;
   private String firstName;
   private int age;
   //-----------------------------------------------------------
   public Person(String last, String first, int a)
      {                               // Конструктор
      lastName = last;
      firstName = first;
      age = a;
      }
   //-----------------------------------------------------------
   public void displayPerson()
      {
      System.out.print("   Last name: " + lastName);
      System.out.print(", First name: " + firstName);
      System.out.println(", Age: " + age);
      }
   //-----------------------------------------------------------
   public String getLast()           // Получение фамилии
      { return lastName; }
   }  // Конец класса Person
////////////////////////////////////////////////////////////////
class ArrayInOb
   {
   private Person[] a;               // Ссылка на массив a
```

```
   private int nElems;               // Количество элементов данных
//--------------------------------------------------------------
   public ArrayInOb(int max)         // Конструктор
      {
      a = new Person[max];           // Создание массива
      nElems = 0;                    // Пока нет ни одного элемента
      }
//--------------------------------------------------------------
                                      // Включение записи в массив
   public void insert(String last, String first, int age)
      {
      a[nElems] = new Person(last, first, age);
      nElems++;                          // Увеличение размера
      }
//--------------------------------------------------------------
   public void display()                // Вывод содержимого массива
      {
      for(int j=0; j<nElems; j++)       // Для каждого элемента
         a[j].displayPerson();          // Вывод
      System.out.println("");
   }
//--------------------------------------------------------------
   public void insertionSort()
      {
      int in, out;

      for(out=1; out<nElems; out++)  // out - разделительный маркер
         {
         Person temp = a[out];       // Скопировать помеченный элемент
         in = out;                   // Начать перемещения с out

         while(in>0 &&               // Пока не найден меньший элемент
               a[in-1].getLast().compareTo(temp.getLast())>0)
            {
            a[in] = a[in-1];         // Сдвинуть элемент вправо
            --in;                    // Перейти на одну позицию влево
            }
         a[in] = temp;               // Вставить помеченный элемент
         }
      }
//--------------------------------------------------------------
   }  // Конец класса ArrayInOb
////////////////////////////////////////////////////////////////
class ObjectSortApp
   {
   public static void main(String[] args)
      {
      int maxSize = 100;             // Размер массива
      ArrayInOb arr;                 // Ссылка на массив
```

продолжение ⇨

Листинг 3.4 (*продолжение*)

```
      arr = new ArrayInOb(maxSize);  // Создание массива

      arr.insert("Evans", "Patty", 24);
      arr.insert("Smith", "Lorraine", 37);
      arr.insert("Yee", "Tom", 43);
      arr.insert("Adams", "Henry", 63);
      arr.insert("Hashimoto", "Sato", 21);
      arr.insert("Stimson", "Henry", 29);
      arr.insert("Velasquez", "Jose", 72);
      arr.insert("Lamarque", "Henry", 54);
      arr.insert("Vang", "Minh", 22);
      arr.insert("Creswell", "Lucinda", 18);

      System.out.println("Before sorting:");
      arr.display();                // Вывод содержимого
      arr.insertionSort();          // Сортировка методом вставки

      System.out.println("After sorting:");
      arr.display();                // Повторный вывод
      }
   }  // Конец класса ObjectSortApp
////////////////////////////////////////////////////////////////
```

Результат выполнения программы:

```
Before sorting:
   Last name: Evans, First name: Patty, Age: 24
   Last name: Smith, First name: Doc, Age: 59
   Last name: Smith, First name: Lorraine, Age: 37
   Last name: Smith, First name: Paul, Age: 37
   Last name: Yee, First name: Tom, Age: 43
   Last name: Hashimoto, First name: Sato, Age: 21
   Last name: Stimson, First name: Henry, Age: 29
   Last name: Velasquez, First name: Jose, Age: 72
   Last name: Vang, First name: Minh, Age: 22
   Last name: Creswell, First name: Lucinda, Age: 18
After sorting:
   Last name: Creswell, First name: Lucinda, Age: 18
   Last name: Evans, First name: Patty, Age: 24
   Last name: Hashimoto, First name: Sato, Age: 21
   Last name: Smith, First name: Doc, Age: 59
   Last name: Smith, First name: Lorraine, Age: 37
   Last name: Smith, First name: Paul, Age: 37
   Last name: Stimson, First name: Henry, Age: 29
   Last name: Vang, First name: Minh, Age: 22
   Last name: Velasquez, First name: Jose, Age: 72
   Last name: Yee, First name: Tom, Age: 43
```

Лексикографические сравнения

Метод `insertSort()` в объекте `objectSort.java` сходен с аналогичным методом программы `insertSort.java`, но он был адаптирован для сравнения объектов по ключевому полю `lastName` вместо значения примитивного типа.

Метод `compareTo()` класса `String` используется для выполнения сравнений в методе `insertSort()`. Выражение, в котором он используется, выглядит так:

```
a[in-1].getLast().compareTo(temp.getLast()) > 0
```

Метод `compareTo()` возвращает разные целочисленные значения в зависимости от относительного лексикографического (то есть алфавитного) расположения объекта `String`, для которого вызван метод, и объекта `String`, переданного в аргументе. Возвращаемые значения перечислены в табл. 3.1.

Например, если `s1` содержит строку «`cat`», а `s2` — строку «`dog`», функция вернет отрицательное число. В программе `objectSort.java` этот метод используется для сравнения строки фамилии `a[in-1]` со строкой фамилии `temp`.

Таблица 3.1. Работа метода compareTo()

s2.compareTo(s1)	**Возвращаемое значение**
s1 < s2	<0
s1 = s2	0
s1 > s2	>0

Устойчивость сортировки

В некоторых случаях важен относительный порядок элементов данных, имеющих одинаковые ключи. Например, данные работников могут быть упорядочены в алфавитном порядке фамилий. (То есть поле фамилии используется в качестве ключа сортировки.) Затем данные потребовалось отсортировать по почтовым индексам, но так, чтобы все элементы с одинаковым индексом остались отсортированными по фамилиям. Алгоритм должен сформировать новые группы, однако в полученных группах должен сохраниться исходный порядок следования элементов. Некоторые алгоритмы сортировки сохраняют этот вторичный порядок; они называются *устойчивыми* (или *стабильными*).

Все алгоритмы, представленные в этой главе, устойчивы. Например, обратите внимание на вывод программы `objectSort.java` (листинг 3.4). В нем присутствуют три человека с фамилией `Smith`. После сортировки сохраняется их исходный относительный порядок следования (`Doc Smith`, `Lorraine Smith`, `Paul Smith`), несмотря на то что эти объекты переместились в новые позиции.

Сравнение простых алгоритмов сортировки

Вероятно, пузырьковую сортировку лучше не использовать, разве что если под рукой у вас не оказалось описаний алгоритмов. Пузырьковая сортировка настолько проста, что ее вполне можно написать «по памяти». Но даже в этом случае она применима только для относительно небольших объемов данных. (О том, что можно считать «небольшим» объемом данных, рассказано в главе 15, «Рекомендации по использованию».)

Сортировка методом выбора сводит к минимуму количество перестановок, но количество сравнений все равно остается высоким. Такая сортировка может пригодиться, если объем данных относительно невелик, а перестановка выполняется намного медленнее сравнения.

Сортировка методом вставки — самый универсальный алгоритм среди тех описанных. Он лучше всего подойдет в большинстве ситуаций, когда объем данных невелик или данные почти отсортированы. Для больших объемов данных самым эффективным обычно считается алгоритм быстрой сортировки; он будет рассмотрен в главе 7.

Мы сравнивали алгоритмы сортировки по скорости обработки. Другим важным фактором для любого алгоритма являются затраты памяти. Все три алгоритма этой главы выполняют сортировку «на месте», иначе говоря, они требуют минимальных дополнительных затрат памяти за пределами исходного массива. Во всех видах сортировки используется дополнительная переменная для временного хранения элемента в ходе перестановки.

Примеры программ (например, `bubbleSort.java`) можно перекомпилировать для обработки больших объемов данных. Хронометраж поможет получить представление о различиях между алгоритмами и времени, необходимом для сортировки разных объемов данных в вашей конкретной системе.

Итоги

- Во всех алгоритмах сортировки, описанных в этой главе, данные хранятся в массиве.
- Сортировка основана на сравнении ключей элементов данных в массиве и перемещении элементов (а вернее, ссылок на них), пока они не будут располагаться в нужном порядке.
- Все алгоритмы этой главы выполняются за время $O(N^2)$. Тем не менее некоторые из них могут работать заметно быстрее других.
- Инвариантом называется условие, которое остается неизменным на протяжении всего времени работы алгоритма.
- Пузырьковая сортировка — наименее эффективный, но самый простой алгоритм.
- Из всех видов сортировки со сложностью $O(N^2)$, описанных в этой главе, чаще всего используется сортировка методом вставки.

- ✦ Сортировка называется устойчивой, если она сохраняет относительный порядок элементов с одинаковыми значениями ключа.
- ✦ Ни один из алгоритмов сортировки этой главы не требует дополнительных затрат памяти за пределами исходного массива (только одна временная переменная).

Вопросы

Следующие вопросы помогут читателю проверить качество усвоения материала. Ответы на них приведены в приложении В.

1. В каком отношении компьютерные алгоритмы сортировки более ограничены по сравнению с сортировкой, выполняемой человеком?
   a) Люди лучше справляются с изобретением новых алгоритмов.
   b) Компьютер может работать с данными фиксированного размера.
   c) Человек знает, как сортировать, а компьютеру необходимо все объяснять.
   d) Компьютер за один раз может сравнивать только два объекта.
2. Две основные операции простой сортировки — __________ элементов и их __________ (иногда __________).
3. При пузырьковой сортировке каждый элемент всегда сравнивается с каждым другим элементом (Да/Нет).
4. В алгоритме пузырьковой сортировки чередуются операции:
   a) сравнения и перестановки;
   b) перемещения и копирования;
   c) перемещения и сравнения;
   d) копирования и сравнения.
5. При N элементах алгоритм пузырьковой сортировки выполняет ровно $N \times N$ сравнений (Да/Нет).
6. В алгоритме сортировки методом выбора:
   a) элементы с наибольшими значениями ключа накапливаются слева (малые значения индексов);
   b) многократно находится значение минимального ключа;
   c) для правильной вставки каждого элемента приходится сдвигать несколько элементов;
   d) отсортированные элементы накапливаются справа.
7. Если при выполнении конкретной сортировки перестановка элементов занимает намного больше времени, чем сравнение, сортировка методом выбора работает примерно вдвое быстрее пузырьковой сортировки (Да/Нет).
8. Копирование выполняется в __________ раза быстрее перестановки.
9. Какой инвариант действует при сортировке методом выбора?

10. Какой переменной программы `insertSort.java` соответствует «помеченный игрок», упоминаемый в тексте при описании сортировки методом вставки?

    a) in;

    b) out;

    c) temp;

    d) a[out].

11. В сортировке методом вставки термин «частичная сортировка» означает, что:

    a) некоторые элементы уже отсортированы, но, возможно, их еще придется перемещать;

    b) большинство элементов находится в своих окончательных позициях сортировки, но некоторые из них еще требуют выполнения сортировки;

    c) отсортированы только некоторые из элементов;

    d) элементы группы отсортированы между собой, но возможно, в группу еще придется вставлять элементы, находящиеся за ее пределами.

12. Сдвиг группы элементов влево или вправо требует многократного выполнения ___________.

13. В сортировке методом вставки элемент, вставленный в частично отсортированную группу:

    a) не будет перемещаться в дальнейшем;

    b) не будет сдвигаться влево;

    c) часто будет перемещаться за пределы группы;

    d) столкнется с тем, что размер группы неуклонно уменьшается.

14. Инвариант сортировки методом вставки: _________.

15. Свойство устойчивости алгоритма сортировки означает, что:

    a) элементы с вторичными ключами исключаются из сортировки;

    b) при сортировке списка городов по штату сохраняется сортировка городов по возрастанию населения;

    c) фамилии ассоциируются с теми же именами;

    d) элементы сохраняют постоянный порядок первичных ключей независимо от порядка вторичных ключей.

Упражнения

Упражнения помогут вам глубже усвоить материал этой главы. Программирования они не требуют.

1. В программе `bubbleSort.java` (листинг 3.1) перепишите метод `main()` так, чтобы он создавал большой массив и заполнял его данными. Для генерирования случайных чисел можно воспользоваться следующим фрагментом кода:

```
for(int j=0; j<maxSize; j++) // Заполнение массива
{ // случайными числами
long n = (long)( java.lang.Math.random()*(maxSize-1) );
arr.insert(n);
}
```

Попробуйте вставить 10 000 чисел. Выведите содержимое массива до и после сортировки. Закомментируйте вызовы `display()` и посмотрите, сколько времени занимает сама сортировка. Ее продолжительность будет зависеть от производительности компьютера, но сортировка 100 000 чисел, вероятно, займет менее 30 секунд. Выберите размер массива, для которого сортировка занимает приблизительно указанное время, и зарегистрируйте его. Затем используйте тот же размер массива в программах `selectSort.java` (листинг 3.2) и `insertSort.java` (листинг 3.3). Сравните скорости выполнения этих сортировок.

2. Напишите код для вставки данных, отсортированных в обратном порядке (99 999, 99 998, 99,997, ...), в программу `bubbleSort.java`. Используйте такой же объем данных, как в упражнении 1. Повторите эксперимент с программами `selectSort.java` и `insertSort.java`.
3. Напишите код для вставки отсортированных данных (0, 1, 2, ...) в программу `bubbleSort.java`. Сравните скорость сортировки с упражнениями 1 и 2. Повторите эксперимент с программами `selectSort.java` и `insertSort.java`.

Программные проекты

В этом разделе читателю предлагаются программные проекты, которые укрепляют понимание материала и демонстрируют практическое применение концепций этой главы.

3.1. В программе `bubbleSort.java` (листинг 3.1) и в приложении BubbleSort Workshop индекс `in` всегда перемещается слева направо, находит наибольший элемент и перемещает его к позиции `out` справа. Измените метод `bubbleSort()` так, чтобы в нем выполнялись двусторонние перемещения, иначе говоря, индекс `in` сначала, как и прежде, переносит наибольший элемент слева направо, но затем он меняет направление и переносит наименьший элемент справа налево. Вам понадобятся два внешних индекса: справа (старый индекс out) и слева.

3.2. Добавьте в класс `ArrayIns` программы `insertSort.java` (листинг 3.3) метод с именем `median()`, возвращающий медиану массива. (Напомним, что в группе чисел половина меньше медианы, а другая половина больше.) Найдите простое решение этой задачи.

3.3. Добавьте в программу `insertSort.java` (листинг 3.3) метод `noDups()`, который удаляет дубликаты из ранее отсортированного массива без нарушения порядка элементов. (Используйте метод `insertionSort()` для сортировки данных или просто вставьте данные в порядке сортировки в `main()`.) Нетрудно представить себе схему, в которой все элементы от позиции обнаружения дубликата до конца массива сдви-

гаются на одну позицию, но это замедлит работу алгоритма до времени $O(N^2)$ — по крайней мере при большом количестве дубликатов. Проследите за тем, чтобы в вашем алгоритме ни один элемент не перемещался более одного раза независимо от количества дубликатов — это обеспечит выполнение алгоритма за время $O(N)$.

3.4. Еще один простой алгоритм сортировки — сортировка методом четно-нечетных перестановок — основан на многократном выполнении двух проходов по массиву. На первом проходе ищутся все пары элементов `a[j]` и `a[j+1]`, где `j` — нечетное число (`j` = 1, 3, 5, ...). Если ключи следуют в неверном порядке, элементы меняются местами. На втором проходе то же самое делается для всех четных значений (`j` = 2, 4, 6, ...). Двухпроходная обработка выполняется многократно до тех пор, пока массив не будет полностью отсортирован. Замените метод `bubbleSort()` в `bubbleSort.java` (листинг 3.1) методом четно-нечетных перестановок `oddEvenSort()`. Убедитесь в том, что он работает для произвольных объемов данных. Требуется определить, сколько раз будет выполняться двухпроходная обработка.

Сортировка методом четно-нечетных перестановок очень полезна в многопроцессорных конфигурациях, когда разные процессоры могут одновременно работать с разными нечетными (а затем и четными) парами. Так как нечетные пары независимы друг от друга, каждая пара может проверяться (с перестановкой элементов в случае необходимости) отдельным процессором. Такая сортировка выполняется очень быстро.

3.5. Измените метод `insertionSort()` в программе `insertSort.java` (листинг 3.3), чтобы он подсчитывал количество копирований и сравнений в ходе сортировки, а затем выводил полученные результаты. Для подсчета сравнений необходимо разбить надвое сложное условие во внутреннем цикле `while`. Используйте программу для измерения количества копирований и сравнения для разных объемов данных, отсортированных в обратном порядке. Подтверждают ли результаты теоретическую сложность $O(N^2)$? Проделайте то же самое для почти отсортированных данных (в которых только несколько элементов находятся не на своих местах). Какие выводы можно сделать об эффективности этого алгоритма для почти отсортированных данных?

3.6. Существует один интересный способ удаления дубликатов из массива. Сортировка методом вставки использует алгоритм с вложенными циклами, который сравнивает каждый элемент массива с каждым другим элементом. Если вы хотите удалить дубликаты, это одно из возможных решений (также см. упражнение 2.6 в главе 2). Измените метод `insertionSort()` в программе `insertSort.java`, чтобы он удалял дубликаты в ходе сортировки. Например, при обнаружении дубликата можно заменить один из экземпляров ключевым значением, которое заведомо меньше ключей других элементов (скажем, –1, если все нормальные ключи положительны). Затем обычный алгоритм сортировки методом вставки, рассматривающий новый ключ наравне со всеми остальными, размещает его в элементе с индексом 0. В дальнейшем он будет игнорироваться алгоритмом. Следующий дубликат размещается в ячейке с индексом 1 и т. д. После завершения сортировки все удаленные дубликаты (теперь представленные ключом –1) будут располагаться в начале массива. Остается лишь сдвинуть недублированные элементы, чтобы они начинались с индекса 0, и изменить размеры массива.

Глава 4

Стеки и очереди

В этой главе рассматриваются три структуры данных: стек, очередь и приоритетная очередь. Сначала будут описаны основные отличия этих структур от массивов, а затем мы рассмотрим каждую структуру по отдельности. Последний раздел посвящен разбору арифметических выражений — области, в которых стек играет особенно важную роль.

Другие структуры

Между структурами данных и алгоритмами, представленными в предыдущих главах, и теми, которые мы будем рассматривать сейчас, существуют значительные различия. Рассмотрим некоторые из них, прежде чем переходить к подробному анализу новых структур.

Инструменты программиста

Массивы, которые рассматривались ранее, а также многие другие структуры, которые встретятся нам позднее (связанные списки, деревья и т. д.), хорошо подходят для хранения информации, типичной для приложений баз данных. Они часто используются для хранения картотек персонала, данных складского учета, финансовых данных и т. д. — данных, представляющих объекты или операции реального мира. Эти структуры обеспечивают удобный доступ к данным: они упрощают вставку, удаление и поиск конкретных элементов.

С другой стороны, структуры и алгоритмы, описанные в этой главе, чаще используются в инструментарии программиста. Они предназначены скорее для упрощения программирования на концептуальном уровне, а не для полноценного хранения данных. Их жизненный цикл обычно короче, чем у структур баз данных. Они создаются и используются для выполнения конкретных задач во время работы программы; когда задача выполнена, эти структуры уничтожаются.

Ограничение доступа

В массиве возможен произвольный доступ к любому элементу — либо напрямую (если известен его индекс), либо поиском по последовательности ячеек, пока нужный элемент не будет найден. Напротив, структуры данных этой главы ограничи-

вают доступ к элементам: в любой момент времени можно прочитать или удалить только один элемент (если действовать по правилам, конечно).

Интерфейс этих структур проектируется с расчетом на поддержку ограничений доступа. Доступ к другим элементам (по крайней мере теоретически) запрещен.

Абстракция

Стеки, очереди и приоритетные очереди являются более абстрактными сущностями, чем массивы и многие другие структуры данных. Они определяются, прежде всего, своим интерфейсом: набором разрешенных операций, которые могут выполняться с ними. Базовый механизм, используемый для их реализации, обычно остается невидимым для пользователя.

Например, как будет показано в этой главе, базовым механизмом для стека может быть массив или связанный список, а базовым механизмом приоритетной очереди — массив или особая разновидность дерева, называемая *кучей*. Мы вернемся к теме реализации структур данных на базе других структур при обсуждении абстрактных типов данных (ADT) в главе 5, «Связанные списки».

Стеки

В стеке доступен только один элемент данных: тот, который был в него вставлен последним. Удалив этот элемент, пользователь получает доступ к предпоследнему элементу и т. д. Такой механизм доступа удобен во многих ситуациях, связанных с программированием. В этой главе будет показано, как использовать стек для проверки сбалансированности круглых, фигурных и угловых скобок в исходном файле компьютерной программы. А в последнем разделе этой главы стек сыграет важнейшую роль в разборе (анализе) арифметических выражений вида $3 \times (4 + 5)$.

Стек также удобен в алгоритмах, применяемых при работе с некоторыми сложными структурами данных. В главе 8, «Двоичные деревья», приведен пример его применения при переборе узлов дерева, а в главе 13, «Графы», стек используется для поиска вершин графа (алгоритм, с помощью которого можно найти выход из лабиринта).

Многие микропроцессоры имеют стековую архитектуру. При вызове метода адрес возврата и аргументы заносятся в стек, а при выходе они извлекаются из стека. Операции со стеком встроены в микропроцессор.

Стековая архитектура также использовалась в некоторых старых калькуляторах. Вместо того чтобы вводить арифметическое выражение с круглыми скобками, пользователь сохранял промежуточные результаты в стеке. Тема будет более подробно описана при обсуждении разбора арифметических выражений в последнем разделе этой главы.

Почтовая аналогия

Для объяснения идеи стека лучше всего воспользоваться аналогией. Многие люди складывают приходящие письма стопкой на журнальном столике. Когда появится свободная минута, они обрабатывают накопившуюся почту сверху вниз. Сначала они открывают письмо, находящееся на вершине стопки, и выполняют необходимое действие — оплачивают счет, выбрасывают письмо и т. д. Разобравшись с первым письмом, они переходят к следующему конверту, который теперь оказывается на верху стопки, и разбираются с ним. В конечном итоге они добираются до нижнего письма (которое теперь оказывается верхним). Стопка писем изображена на рис. 4.1.

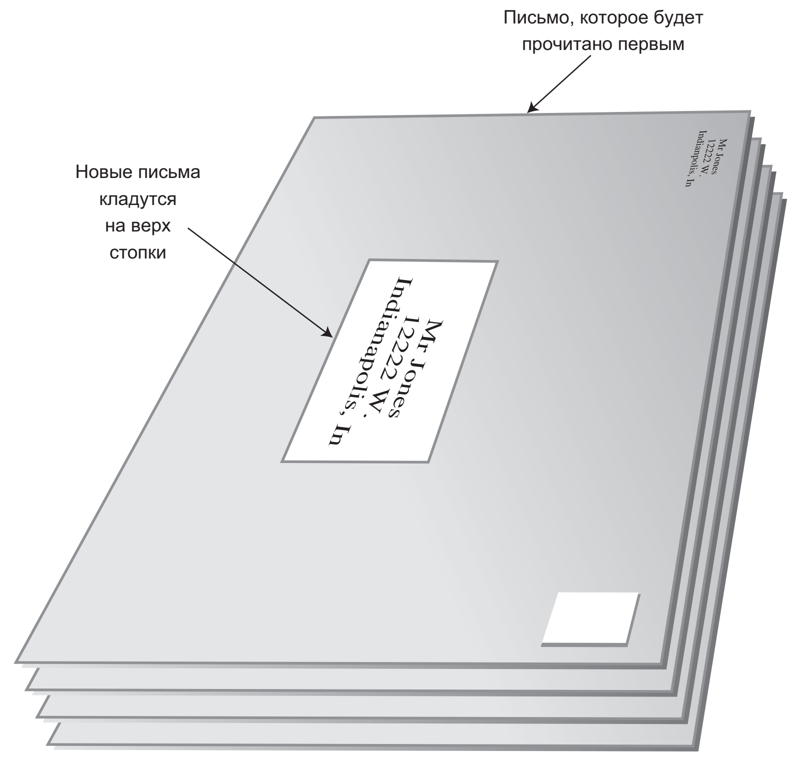


Рис. 4.1. Стопка писем

Принцип «начать с верхнего письма» отлично работает, при условии, что вся почта может быть обработана за разумное время. В противном случае возникает опасность того, что письма в нижней части стопки не будут просматриваться месяцами, а содержащиеся в них счета будут просрочены.

Конечно, не все люди разбирают почту по принципу «сверху вниз». Одни предпочитают брать письма снизу стопки, чтобы старые письма обрабатывались в первую очередь. Другие сортируют почту перед началом обработки и перекладывают важную корреспонденцию наверх. В таких случаях стопка писем уже не является аналогом стека из информатики. Если письма берутся снизу стопки, это

очередь, а если почта сортируется — приоритетная очередь. Обе возможности будут описаны позднее.

Стековую архитектуру также можно сравнить с процессом выполнения различных дел во время рабочего дня. Вы трудитесь над долгосрочным проектом (A), но ваш коллега просит временно прерваться и помочь ему с другим проектом (B). В ходе работы над B к вам заходит бухгалтер, чтобы обсудить ваш отчет по командировочным расходам (C). Во время обсуждения вам срочно звонят из отдела продаж, и вы тратите несколько минут на диагностику своего продукта (D). Разобравшись со звонком D, вы продолжаете обсуждение C; после завершения C возобновляется проект B, а после завершения B вы (наконец-то!) возвращаетесь к проекту A. Проекты с более низкими приоритетами «складываются в стопку», ожидая, пока вы к ним вернетесь.

Основные операции со стеком — *вставка* (занесение) элемента в стек и *извлечение* из стека — выполняются только на вершине стека, то есть с его верхним элементом. Говорят, что стек работает по принципу LIFO (Last-In-First-Out), потому что последний занесенный в стек элемент будет первым извлечен из него.

Приложение Stack Workshop

Приложение Stack Workshop поможет вам лучше понять, как работает стек. В окне приложения находятся четыре кнопки: `New`, `Push`, `Pop` и `Peek` (рис. 4.2).

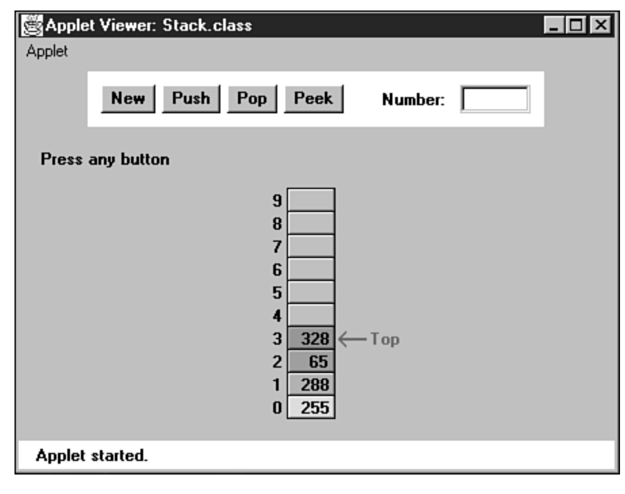


Рис. 4.2. Приложение Stack Workshop

В основу реализации Stack Workshop заложен массив, поэтому в окне выводится ряд ячеек. Однако доступ к стеку ограничивается его вершиной, поэтому вы не сможете обращаться к элементам по индексу. Концепция стека и базовая структура данных, используемая для ее реализации, — совершенно разные понятия. Как упоминалось ранее, стеки также могут реализовываться на базе других структур (например, связанных списков).

Кнопка New

Стек в приложении Stack Workshop изначально содержит четыре элемента. Если вы хотите, чтобы стек создавался пустым, кнопка New создаст стек без элементов. Следующие три кнопки выполняют основные операции со стеком.

Кнопка Push

Кнопка Push вставляет в стек новый элемент данных. После первого нажатия этой кнопки вам будет предложено ввести значение ключа нового элемента. Еще пара щелчков, и вставленный элемент оказывается на вершине стека.

Красная стрелка всегда указывает на вершину стека, то есть на последний вставленный элемент. Обратите внимание на то, как в процессе вставки на одном шаге (нажатии кнопки) смещается вверх указатель Top, а на втором элемент данных собственно вставляется в ячейку. Если бы эти действия выполнялись в обратном порядке, новое значение заменило бы существующий элемент по ссылке Top. При программировании реализации стека важно соблюдать порядок выполнения этих двух операций.

Если стек будет заполнен, то при попытке вставки очередного элемента будет выведено сообщение о ее невозможности. (Теоретически стек на базе ADT переполниться не может, но в реализации на базе массива переполнение не исключено.)

Кнопка Pop

Чтобы извлечь элемент данных с вершины стека, щелкните на кнопке Pop. Извлеченное значение выводится в текстовом поле Number; эта операция соответствует вызову метода pop().

И снова обратите внимание на последовательность действий: сначала элемент извлекается из ячейки, на которую указывает ссылка Top, а затем Top уменьшается и переводится к верхней занятой ячейке. Порядок выполнения этих действий противоположен порядку, используемому при операции Push.

В приложении при нажатии кнопки Pop элемент удаляется из массива, а ячейка окрашивается в серый цвет (признак отсутствия данных). Это не совсем точно — на самом деле удаленные элементы остаются в массиве до тех пор, пока не будут перезаписаны новыми данными. Однако после того, как маркер Top опустится ниже их позиции, эти элементы становятся недоступными, так что на концептуальном уровне они не существуют.

После извлечения из стека последнего элемента маркер Top указывает в позицию –1 под нижней ячейкой. Эта позиция является признаком того, что стек пуст. При попытке извлечь элемент из пустого стека выводится сообщение «Can’t pop: stack is empty».

Кнопка Peek

Вставка и извлечение элементов — две основные операции со стеком. Тем не менее в некоторых ситуациях бывает полезно прочитать значение с вершины стека без его

удаления. Нажав кнопку Peek несколько раз, вы увидите, как значение элемента Top копируется в текстовое поле Number, но сам элемент остается в стеке.

Обратите внимание: «подсмотреть» можно только значение верхнего элемента. Архитектура стека такова, что все остальные элементы остаются невидимыми для пользователя.

Размер стека

Как правило, стек представляет собой небольшую, временную структуру данных; по этой причине мы и показываем стек, состоящий всего из 10 ячеек. Конечно, в реальных программах стек может содержать больше элементов, но на самом деле необходимый размер стека оказывается на удивление незначительным. Например, для разбора очень длинного арифметического выражения обычно хватает стека с 10–12 ячейками.

Реализация стека на языке Java

Программа stack.java реализует стек в виде класса с именем StackX. Листинг 4.1 содержит этот класс и короткий метод main() для его тестирования.

Листинг 4.1. Программа stack.java

```
// stack.java
// Работа со стеком
// Запуск программы: C>java StackApp
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;        // Размер массива
   private long[] stackArray;
   private int top;            // Вершина стека
//--------------------------------------------------------------
   public StackX(int s)         // Конструктор
      {
      maxSize = s;             // Определение размера стека
      stackArray = new long[maxSize];  // Создание массива
      top = -1;                // Пока нет ни одного элемента
      }
//--------------------------------------------------------------
   public void push(long j)      // Размещение элемента на вершине стека
      {
      stackArray[++top] = j;     // Увеличение top, вставка элемента
      }
//--------------------------------------------------------------
   public long pop()             // Извлечение элемента с вершины стека
      {
      return stackArray[top--];  // Извлечение элемента, уменьшение top
      }
```

```
//--------------------------------------------------------------
   public long peek()           // Чтение элемента с вершины стека
      {
      return stackArray[top];
      }
//--------------------------------------------------------------
   public boolean isEmpty()    // True, если стек пуст
      {
      return (top == -1);
      }
//--------------------------------------------------------------
   public boolean isFull()     // True, если стек полон
      {
      return (top == maxSize-1);
      }
//--------------------------------------------------------------
   }  // Конец класса StackX
////////////////////////////////////////////////////////////////
class StackApp
   {
   public static void main(String[] args)
      {
      StackX theStack = new StackX(10);  // Создание нового стека
      theStack.push(20);                 // Занесение элементов в стек
      theStack.push(40);
      theStack.push(60);
      theStack.push(80);

      while( !theStack.isEmpty() )       // Пока стек не станет пустым
         {                               // Удалить элемент из стека
         long value = theStack.pop();
         System.out.print(value);        // Вывод содержимого
         System.out.print(" ");
         }
      System.out.println("");
      }
   }  // Конец класса StackApp
////////////////////////////////////////////////////////////////
```

Метод `main()` класса `StackApp` создает стек для хранения 10 элементов, заносит в него 4 элемента, а затем выводит все элементы, извлекая их из стека, пока он не опустеет. Результат выполнения программы:

```
80 60 40 20
```

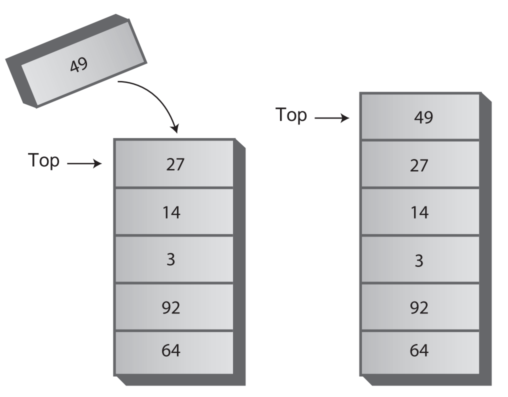
Обратите внимание на обратный порядок данных. Так как последний занесенный элемент стал первым извлеченным элементом, число 80 стоит на первом месте в выходных данных.

Эта версия класса `StackX` хранит элементы данных типа `long`. Как упоминалось в главе 3, «Простая сортировка», их можно заменить любым другим типом, в том числе и объектами.

Методы класса StackX

Конструктор создает новый стек, размер которого передается в аргументе. В полях класса хранится максимальный размер (размер массива), сам массив и переменная `top`, в которой хранится индекс элемента, находящегося на вершине стека. (Обратите внимание: необходимость передачи размера стека объясняется тем, что стек реализован на базе массива. Если бы стек был реализован, допустим, на базе связанного списка, то передавать размер было бы не обязательно.)

Метод `push()` увеличивает `top`, чтобы переменная указывала на ячейку, находящуюся непосредственно над текущей ячейкой, и сохраняет в ней элемент данных. Еще раз обратите внимание: `top` увеличивается *до* вставки элемента.



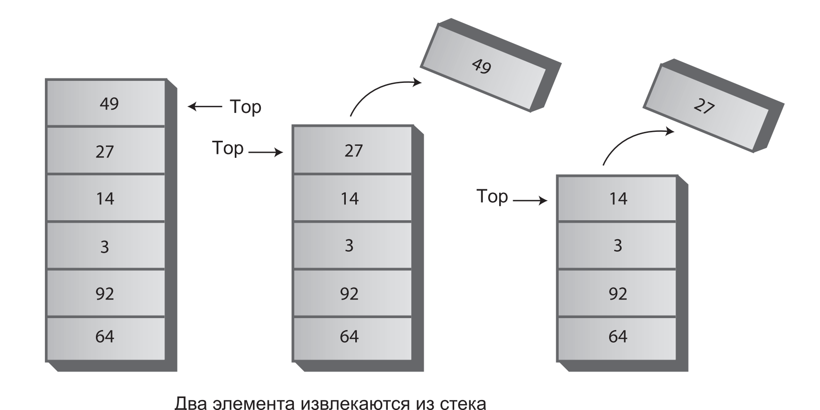


Рис. 4.3. Действие методов класса StackX

Метод `pop()` возвращает значение, находящееся на вершине стека, после чего уменьшает `top`. В результате элемент, находящийся на вершине стека, фактически удаляется; он становится недоступным, хотя само значение остается в массиве (до тех пор, пока в ячейку не будет занесен другой элемент).

Метод `peek()` просто возвращает верхнее значение, не изменяя состояние стека.

Методы `isEmpty()` и `isFull()` возвращают `true`, если стек пуст или полон соответственно. Для пустого стека переменная `top` содержит –1, а для полного — `maxSize-1`.

Рисунок 4.3 показывает, как работают методы класса стека.

Обработка ошибок

Единого подхода к обработке ошибок стека не существует. Например, что должно происходить при занесении элемента в заполненный стек или при попытке извлечения элемента из пустого стека?

Ответственность за обработку таких ошибок возлагается на пользователя класса. Прежде чем вставлять элемент, пользователь должен проверить, остались ли в стеке свободные ячейки:

```
if( !theStack.isFull() )
   insert(item);
else
   System.out.print("Can't insert, stack is full");
```

Ради простоты кода мы исключили проверку из `main()` (к тому же в этой простой программе мы знаем, что стек не заполнен, потому что только что его сами инициализировали). Проверка пустого стека выполняется в методе `main()` при вызове `pop()`. Многие классы стеков выполняют внутреннюю проверку таких ошибок в методах `push()` и `pop()`. Такое решение считается предпочтительным. В Java класс стека, обнаруживший ошибку, обычно инициирует исключение, которое может быть перехвачено и обработано пользователем класса.

Пример использования стека № 1. Перестановка букв в слове

Наш первый пример решает очень простую задачу: перестановку букв в слове. Запустите программу, введите слово и нажмите Enter. Программа выводит слово, в котором буквы переставлены в обратном порядке.

Для перестановки букв используется стек. Сначала символы последовательно извлекаются из входной строки и заносятся в стек, а затем извлекаются из стека и выводятся на экран. Так как стек работает по принципу LIFO, символы извлекаются в порядке, обратном порядку их занесения. Код программы `reverse.java` приведен в листинге 4.2.

Листинг 4.2. Программа reverse.java

```
// reverse.java
// Использование стека для перестановки символов строки
// Запуск программы: C>java ReverseApp
import java.io.*;                 // Для ввода/вывода
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;
```

продолжение ⇨

Листинг 4.2 (*продолжение*)

```
   private char[] stackArray;
   private int top;
//--------------------------------------------------------------
   public StackX(int max)     // Конструктор
      {
      maxSize = max;
      stackArray = new char[maxSize];
      top = -1;
      }
//--------------------------------------------------------------
   public void push(char j)  // Размещение элемента на вершине стека
      {
      stackArray[++top] = j;
      }
//--------------------------------------------------------------
   public char pop()         // Извлечение элемента с вершины стека
      {
      return stackArray[top--];
      }
//--------------------------------------------------------------
   public char peek()        // Чтение элемента с вершины стека
      {
      return stackArray[top];
   }
//--------------------------------------------------------------
   public boolean isEmpty()  // True, если стек пуст
      {
      return (top == -1);
      }
//--------------------------------------------------------------
   }  // Конец класса StackX
////////////////////////////////////////////////////////////////
class Reverser
   {
   private String input;                // Входная строка
   private String output;               // Выходная строка
//--------------------------------------------------------------
   public Reverser(String in)           // Конструктор
      { input = in; }
//--------------------------------------------------------------
   public String doRev()                // Перестановка символов
      {
      int stackSize = input.length();   // Определение размера стека
      StackX theStack = new StackX(stackSize);  // Создание стека

      for(int j=0; j<input.length(); j++)
         {
         char ch = input.charAt(j);     // Чтение символа из входного потока
         theStack.push(ch);             // Занесение в стек
         }
```

```
      output = "";
      while( !theStack.isEmpty() )
         {
         char ch = theStack.pop();      // Извлечение символа из стека
         output = output + ch;          // Присоединение к выходной строке
         }
      return output;
      }
//--------------------------------------------------------------
   }  // Конец класса Reverser
////////////////////////////////////////////////////////////////
class ReverseApp
   {
   public static void main(String[] args) throws IOException
      {
      String input, output;
      while(true)
         {
         System.out.print("Enter a string: ");
         System.out.flush();
         input = getString();          // Чтение строки с клавиатуры
         if( input.equals("") )        // Завершение, если [Enter]
            break;
                                       // Создание объекта Reverser
         Reverser theReverser = new Reverser(input);
         output = theReverser.doRev(); // Использование
         System.out.println("Reversed: " + output);
         }
      }
//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   }  // Конец класса ReverseApp
////////////////////////////////////////////////////////////////
```

Класс Reverser переставляет символы своей входной строки. Его важнейший компонент — метод doRev() — выбирает размер стека в соответствии с длиной входной строки.

В методе main() мы получаем строку от пользователя, создаем объект Reverser с передачей этой строки в аргументе конструктора, вызываем метод doRev() объекта и выводим возвращаемое значение, в котором и содержится строка с переставленными символами. Пример взаимодействия с программой:

```
Enter a string: part
Reversed: trap
Enter a string:
```

Пример № 2. Поиск парных скобок

Стеки также часто используются при разборе некоторых видов текстовых строк. Как правило, строки содержат программный код, написанный на компьютерном языке, и обрабатываются программой-компилятором.

Чтобы дать представление о процессе разбора, мы рассмотрим программу для проверки парных скобок в строке, введенной пользователем. Строка не обязана содержать настоящий Java-код (хотя это вполне возможно), но скобки в ней должны использоваться по правилам Java. Нас интересуют фигурные скобки { }, квадратные [ ] и круглые скобки ( ). Каждая открывающая (левая) скобка должна иметь парную закрывающую (правую) скобку, то есть для каждой скобки { должна существовать парная скобка } и т. д. Кроме того, открывающие скобки ближе к концу строки закрываются раньше тех, которые расположены ближе к началу. Несколько примеров:

```
c[d]        // Правильно
a{b[c]d}e   // Правильно
a{b(c]d}e   // Неправильно; ] не соответствует (
a[b{c}d]e}  // Неправильно; у завершающей скобки } нет пары
a{b(c)      // Неправильно; у открывающей скобки { нет пары
```

Открывающие скобки в стеке

Программа поиска парных скобок последовательно читает символы строки и заносит обнаруженные открывающие скобки в стек. Обнаружив во входных данных закрывающую скобку, она извлекает верхний элемент из стека и проверяет его на соответствие закрывающей скобке. Если они относятся к разным типам (скажем, открывающая фигурная скобка с закрывающей круглой скобкой), происходит ошибка. Кроме того, если в стеке нет открывающей скобки, парной по отношению к закрывающей, или если для какой-то скобки в конечном итоге не нашлось пары, происходит ошибка. Найти непарный ограничитель несложно — он остается в стеке после того, как будут прочитаны все символы в строке.

Давайте посмотрим, как будет заполняться стек для типичной синтаксически правильной строки:

```
a{b(c[d]e)f}
```

В таблице 4.1 показано, как выглядит стек при чтении очередного символа из строки. Во втором столбце приводится содержимое стека (вершина справа).

В процессе чтения строки каждая открывающая скобка помещается в стек. Каждая закрывающая скобка, прочитанная из входных данных, сопоставляется с открывающей скобкой, извлеченной с вершины стека. Если они образуют пару — все хорошо. В стек заносятся только скобки; все остальные символы просто игнорируются.

Такое решение работает, так как пара скобок, открытая последней, будет закрыта первой — стоит напомнить, что стек работает по принципу LIFO.

Таблица 4.1. Содержимое стека при поиске парных скобок

Прочитанный символ	Содержимое стека
a	
{	{
b	{
(	{(
c	{(
[	{([
d	{([
]	{(
e	{(
)	{
f	{
}	

Реализация brackets.java на языке Java

Программа разбора скобок `brackets.java` приведена в листинге 4.3. Поиск парных скобок выполняется методом `check()` класса `BracketChecker`.

Листинг 4.3. Программа brackets.java

```
// brackets.java
// Использование стека для поиска парных скобок
// Запуск программы: C>java BracketsApp
import java.io.*;                 // Для ввода/вывода
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;
   private char[] stackArray;
   private int top;
//--------------------------------------------------------------
   public StackX(int s)       // Конструктор
      {
      maxSize = s;
      stackArray = new char[maxSize];
      top = -1;
      }
//--------------------------------------------------------------
   public void push(char j)  // Размещение элемента на вершине стека
      {
      stackArray[++top] = j;
      }
```

продолжение ☞

Листинг 4.3 (*продолжение*)

```
//--------------------------------------------------------------
   public char pop()         // Извлечение элемента с вершины стека
      {
      return stackArray[top--];
      }
//--------------------------------------------------------------
   public char peek()        // Чтение элемента с вершины стека
      {
      return stackArray[top];
      }
//--------------------------------------------------------------
   public boolean isEmpty()    // True, если стек пуст
      {
      return (top == -1);
      }
//--------------------------------------------------------------
   }  // Конец класса StackX
////////////////////////////////////////////////////////////////
class BracketChecker
{
   private String input;                   // Входная строка
//--------------------------------------------------------------
   public BracketChecker(String in)        // Конструктор
      { input = in; }
//--------------------------------------------------------------
   public void check()
      {
      int stackSize = input.length();      // Определение размера стека
      StackX theStack = new StackX(stackSize);  // Создание стека

      for(int j=0; j<input.length(); j++)  // Последовательное чтение
         {
         char ch = input.charAt(j);        // Чтение символа
         switch(ch)
            {
            case ‘{‘:                      // Открывающие скобки
            case '[':
            case '(':
               theStack.push(ch);          // Занести в стек
               break;

            case ‘}’:                      // Закрывающие скобки
            case ']':
            case ')':
               if( !theStack.isEmpty() )   // Если стек не пуст,
                  {
                  char chx = theStack.pop();  // Извлечь и проверить
                  if( (ch=='}’ && chx!='{‘) ||
                      (ch==']' && chx!='[') ||
```

```
                    (ch==')' && chx!='(') )
                   System.out.println("Error: "+ch+" at "+j);
                }
             else              // Преждевременная нехватка элементов
                System.out.println("Error: "+ch+" at "+j);
             break;
          default:    // Для других символов действия не выполняются
             break;
          }
       }
    // В этой точке обработаны все символы
    if( !theStack.isEmpty() )
       System.out.println("Error: missing right delimiter");
}//-------------------------------------------------------------
   }  // Конец класса BracketChecker
////////////////////////////////////////////////////////////////
class BracketsApp
   {
   public static void main(String[] args) throws IOException
      {
      String input;
      while(true)
         {
         System.out.print(
            "Enter string containing delimiters: ");
         System.out.flush();
         input = getString();     // Чтение строки с клавиатуры
         if( input.equals("") )   // Завершение, если [Enter]
            break;
                                  // Создание объекта BracketChecker
         BracketChecker theChecker = new BracketChecker(input);
         theChecker.check();      // Проверка парных скобок
         }
      }
//-------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//-------------------------------------------------------------
   }  // Конец класса BracketsApp
////////////////////////////////////////////////////////////////
```

Метод check() использует класс StackX из программы reverse.java (см. листинг 4.2). Обратите внимание, насколько легко повторно использовать этот класс: весь необходимый код находится в одном месте. Это одно из преимуществ объектно-ориентированного программирования.

Метод main() класса BracketsApp в цикле запрашивает у пользователя строку текста, создает объект BracketChecker, передавая полученную строку в аргументе, после чего вызывает метод check() объекта BracketChecker. Если в ходе разбора будут обнаружены ошибки, метод check() выводит информацию о них; в противном случае синтаксис парных скобок правилен.

Метод check() по возможности сообщает номер символа, в котором была обнаружена ошибка (начиная с 0 для крайней левой позиции) и обнаруженный неправильный символ. Например, для входной строки

```
a{b(c]d}e
```

метод check() выведет следующую информацию:

```
Error: ] at 5
```

Стек как инструмент программиста

Посмотрите, как удобно было использовать стек в программе brackets.java. Можно было создать массив, который делал бы то же самое, но тогда пришлось бы отслеживать индекс последнего добавленного символа, а также хранить другую служебную информацию. Стек в нашей задаче более удобен на концептуальном уровне. Благодаря ограничению доступа к элементам и использованию методов push()/pop() стек делает программу более понятной и снижает риск ошибок. (Как вам скажет любой плотник, самый подходящий инструмент наиболее безопасен.)

Эффективность стеков

Занесение и извлечение элементов из стека, реализованного в классе StackX, выполняется за время $O(1)$. Иначе говоря, время выполнения операции не зависит от количества элементов в стеке; следовательно, операция выполняется очень быстро, не требуя ни сравнений, ни перемещений.

Очереди

Структура данных, называемая в информатике *очередью*, напоминает стек, но в очереди первым извлекается элемент, вставленный первым (FIFO, First-In-First-Out), тогда как в стеке, как мы видели, первым извлекается элемент, вставленный последним (LIFO). Она работает по тому же принципу, что и очередь в кино: человек, первым вставшим в очередь, первым доберется до кассы и купит билет. Тот, кто встанет в очередь последним, последним купит билет (или не купит, если билеты будут распроданы). На рис. 4.4 показано, как работает очередь.

Очередь — такой же вспомогательный инструмент программиста, как и стек. В главе 13 будет рассмотрен пример использования очереди для поиска по графу. Кроме того, очереди используются для моделирования реальных ситуаций ожидания: клиентов в банке, вылета самолетов или передачи данных по Интернету.



Рис. 4.4. Очередь

В операционной системе вашего компьютера (и в сети) трудятся различные очереди, незаметно выполняющие свои обязанности. В очереди печати задания ждут освобождения принтера. Данные, вводимые с клавиатуры, тоже сохраняются в очереди. Если вы работаете в текстовом редакторе, а компьютер на короткое время отвлекся на выполнение другой операции, нажатия клавиш не будут потеряны; они ожидают в очереди, пока у редактора не появится свободное время для их получения. Очередь обеспечивает хранение нажатий клавиш в исходной последовательности до момента обработки.

Приложение Queue Workshop

Приложение Queue Workshop дает представление о том, как работают очереди. При запуске приложения в окне отображается очередь с четырьмя заранее созданными элементами (рис. 4.5).

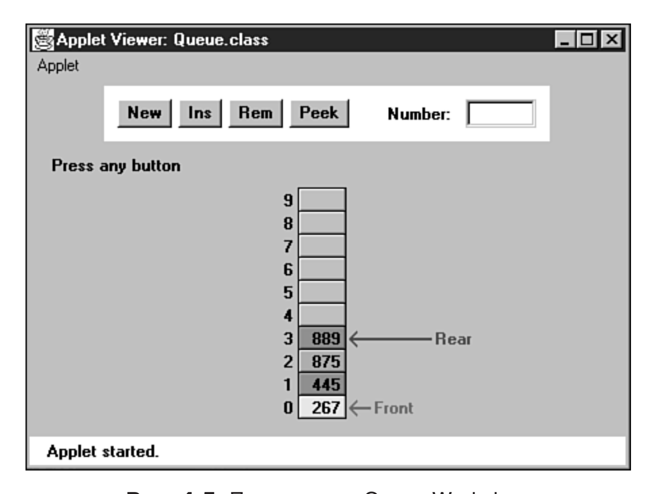


Рис. 4.5. Приложение Queue Workshop

В апплете используется очередь, реализованная на основе массива. Это весьма распространенное решение, хотя очереди также часто реализуются на базе связанных списков.

Две основные операции с очередью — *вставка* элемента в конец очереди и *извлечение* элемента с начала очереди. Все происходит так же, как в очереди в кино: человек становится в конец очереди, ждет, добирается до кассы, покупает билет и покидает очередь из ее начала.

Кнопка Ins

Кнопка `Ins` в приложении Queue Workshop вставляет новый элемент в очередь (операция требует нескольких нажатий кнопки). После первого нажатия вам будет предложено ввести значение нового ключа в текстовом поле `Number`; это должно быть число в диапазоне от 0 до 999. Элемент с указанным ключом вставляется в конец очереди, а маркер `Rear` переходит к новому элементу.

Кнопка Rem

Кнопка `Rem` предназначена для извлечения элемента, находящегося в начале очереди. Значение извлеченного элемента сохраняется в поле `Number` (аналог возвращаемого значения метода `remove()`), а маркер `Front` перемещается к предыдущему элементу. В приложении ячейка, содержащая извлеченный элемент, окрашивается в серый цвет, показывающий, что элемент исчез из очереди. В обычной реализации данные остаются в памяти, но становятся недоступными, потому что указатель `Front` указывает на другой элемент. Операции вставки и извлечения элементов из очереди показаны на рис. 4.6.

В отличие от стека элементы очереди не всегда следуют до ячейки массива с индексом 0. После удаления некоторых элементов `Front` указывает на ячейку с более высоким индексом, как показано на рис. 4.7.

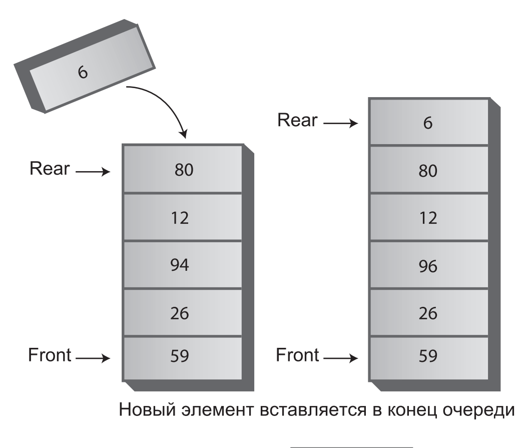
Обратите внимание: на рис. 4.7 маркер начала очереди `Front` находится ниже `Rear`; таким образом, `Front` обладает меньшим индексом. Как вы вскоре убедитесь, это не всегда так.

Кнопка Peek

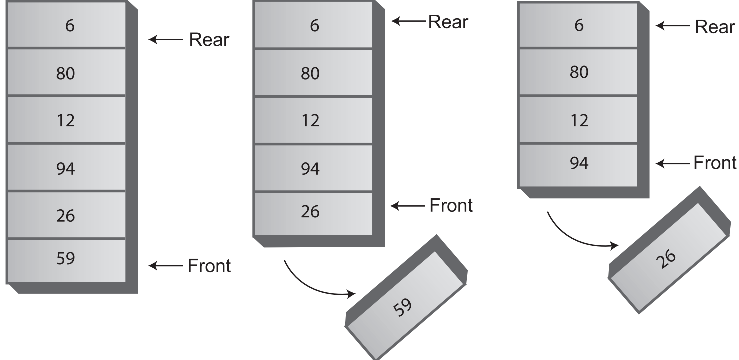
Приложение демонстрирует еще одну операцию с очередью: операция чтения возвращает значение элемента, находящегося в начале очереди, без его удаления. При нажатии кнопки `Peek` значение в ячейке `Front` копируется в поле `Number`, а состояние очереди остается неизменным. Метод `peek()` возвращает значение элемента, находящегося в начале очереди. У некоторых реализаций очередей имеются методы `rearPeek()` и `frontPeek()` (возвращающие значение последнего и первого элемента соответственно), но обычно пользователь желает знать, какой элемент сейчас будет исключен из очереди, а не то, какой элемент был сейчас в нее вставлен.

Кнопка New

Чтобы начать эксперименты с пустой очереди, создайте ее кнопкой `New`.



Новый элемент вставляется в конец очереди



Извлечение двух элементов от начала очереди

Рис. 4.6. Действие методов класса Queue

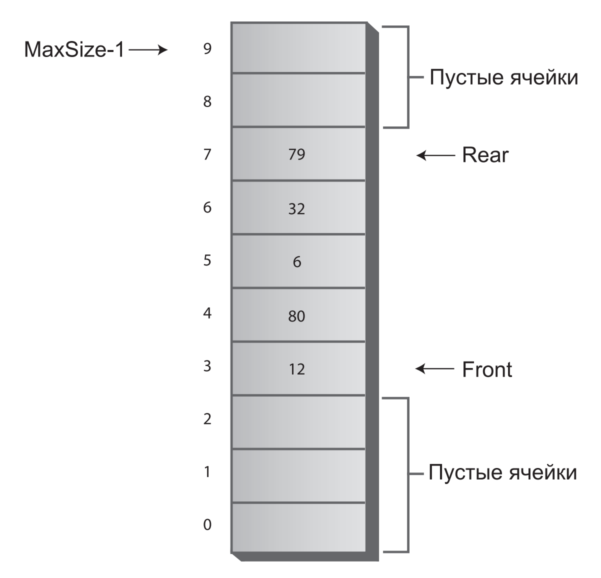


Рис. 4.7. Очередь с удаленными элементами

Пустая и заполненная очередь

При попытке удаления из пустой очереди или вставки в заполненную очередь выводится соответствующее сообщение об ошибке.

Циклическая очередь

При вставке нового элемента в приложении Queue Workshop маркер `Front` смещается вверх, в направлении более высоких индексов. При удалении элемента маркер `Rear` тоже смещается вверх. Попробуйте выполнить эти операции в приложении и убедитесь в том, что это действительно так. На первый взгляд такое поведение выглядит противоестественно — ведь когда вы стоите в очереди в кино, при выходе очередного человека все остальные сдвигаются к началу очереди. Конечно, при удалении можно было бы перемещать все элементы в очереди, но копирование снизило бы эффективность операции. Вместо этого мы оставляем все элементы на своих местах, а смещаем начало и конец очереди.

Проблема в том, что довольно скоро конец очереди достигнет границы массива (элемент с наибольшим индексом). Даже если в начале массива имеются пустые ячейки, из которых были удалены элементы, вставить новый элемент не удастся, потому что маркеру `Rear` дальше двигаться некуда. Или все-таки...? Ситуация изображена на рис. 4.8.

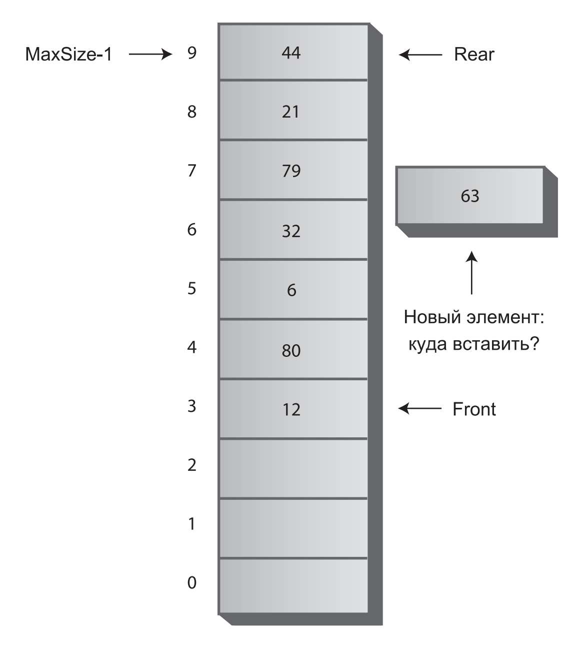


Рис. 4.8. Маркер Rear достиг конца массива

Циклический перенос

Для решения проблемы со вставкой новых элементов в очередь, в которой имеются свободные ячейки, маркеры `Front` и `Rear` при достижении границы массива пере-

мещается к его началу. Такая структура данных называется *циклической очередью* (также иногда встречается термин *кольцевой буфер*).

Приложение Queue Workshop показывает, как работает циклический перенос. Вставляйте в очередь элементы, пока стрелка `Rear` не дойдет до верхнего элемента массива (индекс 9). Удалите несколько элементов (в начале массива) и вставьте очередной элемент. Маркер `Rear` переходит от индекса 9 к индексу 0; новый элемент будет вставлен в этой позиции. Ситуация изображена на рис. 4.9.

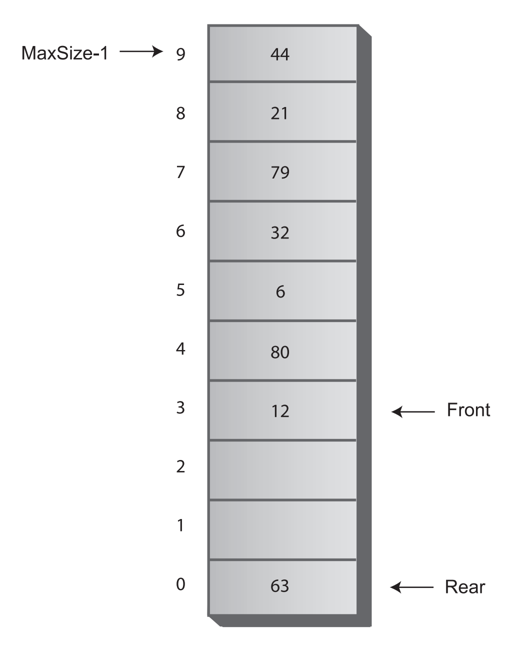


Рис. 4.9. Циклический перенос маркера Rear

Вставьте еще несколько элементов. Маркер `Rear` двигается вверх, как и следовало ожидать. Обратите внимание: после переноса маркер `Rear` оказывается *ниже* маркера `Front` (ситуация, противоположная исходной). Фактически элементы массива образуют две разные последовательности.

Удалите несколько элементов, чтобы маркер `Rear` тоже переместился к началу. Восстанавливается исходная конфигурация, то есть маркер `Front` находится ниже `Rear`. Элементы образуют одну *непрерывную* последовательность.

Реализация очереди на языке Java

Программа `queue.java` реализует очередь в виде класса `Queue`. Класс содержит типичные методы `insert()`, `remove()`, `peek()`, `isFull()`, `isEmpty()` и `size()`.

Программа `main()` создает очередь из пяти ячеек, вставляет в нее четыре элемента, затем удаляет три элемента и вставляет еще четыре. При шестой вставке происходит циклический перенос. Затем программа извлекает и выводит значения всех элементов. Результат выглядит так:

```
40 50 60 70 80
```

Код программы queue.java приведен в листинге 4.4.

Листинг 4.4. Программа queue.java

```
// queue.java
// Работа с очередью
// Запуск программы: C>java QueueApp
////////////////////////////////////////////////////////////////
class Queue
   {
   private int maxSize;
   private long[] queArray;
   private int front;
   private int rear;
   private int nItems;
//--------------------------------------------------------------
   public Queue(int s)          // Конструктор
      {
      maxSize = s;
      queArray = new long[maxSize];
      front = 0;
      rear = -1;
      nItems = 0;
      }
//--------------------------------------------------------------
   public void insert(long j)   // Вставка элемента в конец очереди
      {
      if(rear == maxSize-1)         // Циклический перенос
      rear = -1;
      queArray[++rear] = j;         // Увеличение rear и вставка
      nItems++;                     // Увеличение количества элементов
      }
//--------------------------------------------------------------
   public long remove()         // Извлечение элемента в начале очереди
      {
      long temp = queArray[front++]; // Выборка и увеличение front
      if(front == maxSize)           // Циклический перенос
      front = 0;
      nItems--;                      // Уменьшение количества элементов
      return temp;
      }
//--------------------------------------------------------------
   public long peekFront()      // Чтение элемента в начале очереди
      {
      return queArray[front];
      }
//--------------------------------------------------------------
   public boolean isEmpty()    // true, если очередь пуста
      {
      return (nItems==0);
      }
//--------------------------------------------------------------
   public boolean isFull()     // true, если очередь заполнена
```

```
      {
      return (nItems==maxSize);
      }
//--------------------------------------------------------------
   public int size()          // Количество элементов в очереди
      {
      return nItems;
      }
//--------------------------------------------------------------
   }  // Конец класса Queue
////////////////////////////////////////////////////////////////
class QueueApp
   {
   public static void main(String[] args)
      {
      Queue theQueue = new Queue(5);  // Очередь из 5 ячеек

      theQueue.insert(10);            // Вставка 4 элементов
      theQueue.insert(20);
      theQueue.insert(30);
      theQueue.insert(40);

      theQueue.remove();              // Извлечение 3 элементов
      theQueue.remove();              //    (10, 20, 30)
      theQueue.remove();

      theQueue.insert(50);            // Вставка еще 4 элементов
      theQueue.insert(60);            //    (с циклическим переносом)
      theQueue.insert(70);
      theQueue.insert(80);

      while( !theQueue.isEmpty() )    // Извлечение и вывод
         {                            // всех элементов
         long n = theQueue.remove();  // (40, 50, 60, 70, 80)
         System.out.print(n);
         System.out.print(" ");
         }
      System.out.println("");
      }
   }  // Конец класса QueueApp
```

В нашей реализации в полях класса Queue хранятся не только индексы начала и конца очереди, но и текущее количество элементов nItems. Некоторые реализации очереди не используют это поле; альтернативное решение будет представлено позднее.

Метод insert()

Метод insert() предполагает, что очередь еще не заполнена. В методе main() соответствующей проверки нет, но обычно insert() вызывается только после пред-

варительного вызова isFull() и проверки возвращаемого значения. (Хотя еще правильнее включить проверку в insert() и инициировать исключение при попытке вставки в заполненную очередь.)

В общем случае при вставке сначала увеличивается индекс rear, а затем новый элемент вставляется в ячейку, на которую ссылается его новое значение. Но если значение rear уже достигло вершины массива (maxSize-1), то перед выполнением вставки оно должно вернуться к нижней границе. Для этого полю rear присваивается значение −1, чтобы при увеличении оно стало равным 0 (нижняя граница массива). Код метода завершается увеличением количества элементов nItems.

Метод remove()

Метод remove() предполагает, что очередь не пуста. Вызовите isEmpty() перед remove(), чтобы убедиться в истинности этого условия, или встройте проверку ошибок в remove().

Извлечение всегда начинается с получения значения в ячейке front и увеличения front. Если значение front при этом выходит за границу массива, оно возвращается к 0. На время проверки этой возможности возвращаемое значение сохраняется во временной переменной. Код метода завершается уменьшением количества элементов nItems.

Метод peek()

Метод peek() очень прост: он возвращает значение из ячейки front. Некоторые реализации также позволяют читать значение из нижней границы массива; в таких случаях методам обычно присваиваются имена peekFront() и peekRear() (или просто front() и rear()).

Методы isEmpty(), isFull() и size()

Возвращаемое значение методов isEmpty(), isFull() и size() зависит от поля nItems. Первые два метода сравнивают его с 0 и maxSize, а третий возвращает текущее значение.

Реализация без счетчика элементов

Включение поля nItems в класс Queue слегка снижает эффективность методов insert() и remove(), которым приходится соответственно увеличивать и уменьшать эту переменную. На первый взгляд потери невелики, но при очень большом количестве операций вставки и удаления они могут повлиять на производительность.

По этой причине некоторые реализации очередей обходятся без счетчика элементов, а для проверки заполненности/отсутствия элементов и количества элементов в очереди используются значения полей front и rear. Однако методы isEmpty(), isFull() и size() становятся неожиданно сложными, потому что последовательность элементов, как мы уже видели, может быть как непрерывной, так и несвязной.

Также может возникнуть одна странная проблема: при заполненной очереди указатели `front` и `rear` могут занимать точно такие же позиции, как и при пустой очереди. Очередь выглядит заполненной и пустой одновременно. Для решения этой проблемы создается массив с количеством ячеек, на единицу большим максимального количества элементов, которые в нем будут размещаться. Пример реализации класса очереди без счетчика элементов представлен в листинге 4.5.

Листинг 4.5. Реализация очереди без счетчика элементов

```
class Queue
   {
   private int maxSize;
   private long[] queArray;
   private int front;
   private int rear;
//--------------------------------------------------------------
   public Queue(int s)         // Конструктор
      {
      maxSize = s+1;                  // Массив на одну ячейку больше
      queArray = new long[maxSize];  // требуемого размера
      front = 0;
      rear = -1;
      }
//--------------------------------------------------------------
   public void insert(long j)  // Вставка элемента в конец очереди
      {
      if(rear == maxSize-1)
      rear = -1;
      queArray[++rear] = j;
      }
//--------------------------------------------------------------
   public long remove()        // Извлечение элемента в начале очереди
      {
      long temp = queArray[front++];
      if(front == maxSize)
         front = 0;
      return temp;
      }
//--------------------------------------------------------------
   public long peek()          // Чтение элемента в начале очереди
      {
      return queArray[front];
      }
//--------------------------------------------------------------
   public boolean isEmpty()    // true, если очередь пуста
      {
      return ( rear+1==front || (front+maxSize-1==rear) );
      }
//--------------------------------------------------------------
   public boolean isFull()     // true, если очередь заполнена
```

продолжение ⇨

Листинг 4.5 (*продолжение*)

```
      {
      return ( rear+2==front || (front+maxSize-2==rear) );
      }
//--------------------------------------------------------------
   public int size()          // (assumes queue not empty)
      {
      if(rear >= front)            // Непрерывная последовательность
         return rear-front+1;
      else                         // Несвязная последовательность
         return (maxSize-front) + (rear+1);
   }
//--------------------------------------------------------------
   }  // Конец класса Queue
```

Обратите внимание на усложнение методов `isFull()`, `isEmpty()` и `size()`. На практике реальная необходимость в таком решении встречается очень редко, поэтому мы не будем подробно рассматривать его.

Эффективность очередей

Вставка и извлечение элементов очереди, как и элементов стека, выполняются за время $O(1)$.

Дек

Дек (deque) представляет собой двустороннюю очередь. И вставка, и удаление элементов могут производиться с обоих концов. Соответствующие методы могут называться `insertLeft()/insertRight()` и `removeLeft()/removeRight()`.

Если ограничиться только методами `insertLeft()` и `removeLeft()` (или их эквивалентами для правого конца), дек работает как стек. Если же ограничиться методами `insertLeft()` и `removeRight()` (или противоположной парой), он работает как очередь.

По своей гибкости деки превосходят и стеки, и очереди; иногда они используются в библиотеках классов-контейнеров для реализации обеих разновидностей. Тем не менее используются они реже стеков или очередей, поэтому подробно рассматривать мы их тоже не будем.

Приоритетные очереди

Приоритетная очередь является более специализированной структурой данных, чем стек или очередь, однако и он неожиданно часто оказывается полезным. У приоритетной очереди, как и у обычной, имеется начало и конец, а элементы извлекаются от начала. Но у приоритетной очереди элементы упорядочиваются по ключу, так что элемент с наименьшим (в некоторых реализациях — наибольшим) значением ключа всегда находится в начале. Новые элементы вставляются в позициях, сохраняющих порядок сортировки.

Давайте применим нашу аналогию с сортировкой почты к приоритетным очередям. Каждый раз, когда почтальон отдает вам письмо, вы вкладываете его в стопку непрочитанных писем в соответствии с приоритетом. Если письмо требует немедленного ответа (телефонная компания собирается отключить ваш модем), оно кладется наверх, а письма, на которые можно ответить в свободное время (письмо от любимой тетушки), подкладываются под низ стопки. Письма со средним приоритетом размещаются где-то в середине; чем выше приоритет, тем выше оказывается письмо в стопке. Верх стопки соответствует началу приоритетной очереди.

Когда у вас появляется время на чтение, вы берете письмо с верха стопки (из начала очереди); таким образом, самые важные письма будут обработаны первыми. Ситуация изображена на рис. 4.10.

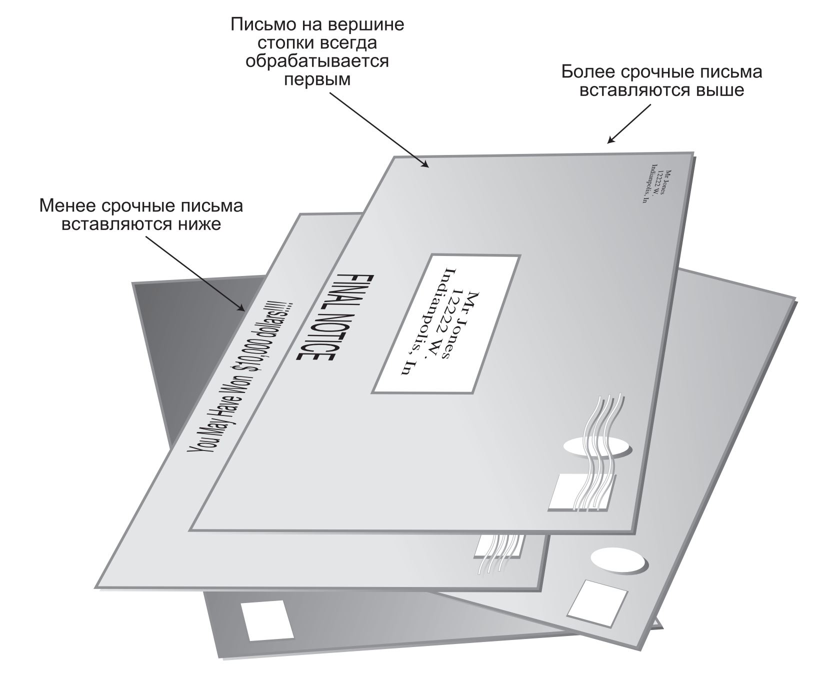


Рис. 4.10. Письма в приоритетной очереди

Приоритетные очереди, как и стеки с очередями, часто используются программистами как вспомогательные инструменты. Пример такого рода встретится нам при построении *связующего дерева* графа в главе 14, «Взвешенные графы».

Кроме того, приоритетные очереди (как и обычные) часто используются в компьютерных системах. Скажем, в операционной системе с вытесняющей многозадачностью программы могут размещаться в приоритетной очереди, чтобы высокоприоритетная программа первой получила процессорное время для ее выполнения.

При работе с приоритетными очередями довольно часто требуется получить доступ к элементу с наименьшим значением ключа (которое может представлять самый экономичный или быстрый способ выполнения какой-либо операции) — то есть самым приоритетным является элемент с наименьшим ключом. Наше обсуж-

дение будет основано именно на таком предположении, хотя во многих ситуациях самым приоритетным является элемент с наибольшим ключом.

Кроме ускоренного доступа к элементу с наименьшим ключом, приоритетная очередь также должна обеспечивать относительно быструю вставку. По этой причине приоритетные очереди, как упоминалось ранее, часто реализуются на основе структуры данных, называемой *кучей* (heap), — эта структура рассматривается в главе 12. А в этой главе будет представлена реализация приоритетной очереди на базе простого массива. Ее недостатком является медленная вставка; с другой стороны, она проще, а ее применение более уместно, если количество элементов невелико или скорость вставки не критична.

Приложение The PriorityQ Workshop

Приложение PriorityQ Workshop реализует приоритетную очередь на базе массива, с хранением всех элементов в порядке сортировки. Очередь упорядочена *по возрастанию приоритетов*, то есть элемент с наименьшим ключом обладает самым высоким приоритетом, и извлекается при вызове `remove()`. (Если бы при вызове происходило обращение к элементу с наибольшим ключом, то очередь была бы упорядочена *по убыванию приоритетов*.)

Элемент с наименьшим значением ключа всегда хранится на вершине массива (в ячейке с наибольшим индексом), а элемент с наименьшим значением ключа — в ячейке с индексом 0. На рис. 4.11 изображено окно приложения в момент запуска. В исходном состоянии очередь содержит пять элементов.

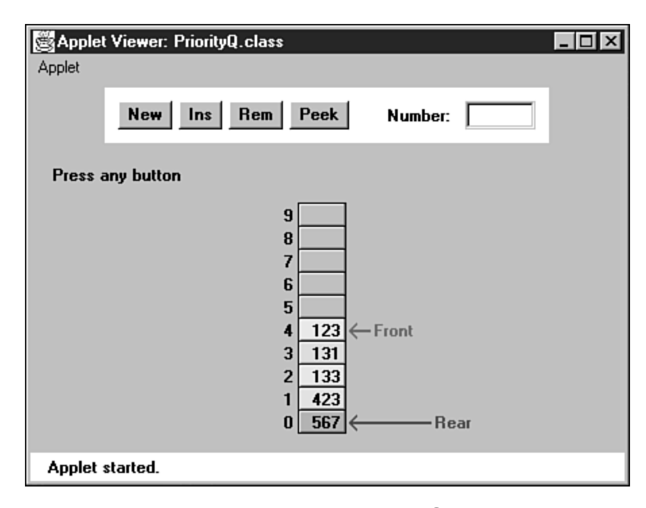


Рис. 4.11. Приложение PriorityQ Workshop

Вставка

Попробуйте вставить новый элемент при помощи кнопки `Ins`. Вам будет предложено ввести значение ключа нового элемента в поле `Number`. Выберите значение, находящееся примерно в середине текущего диапазона ключей очереди. Например, для очереди на рис. 4.11 можно выбрать значение 300. При последующих нажатиях кнопки `Ins` элементы с меньшими значениями ключа сдвигаются вверх,

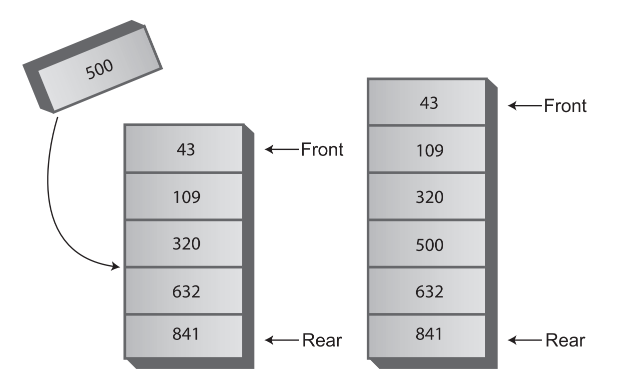
чтобы освободить место для элементов с меньшими ключами. Текущий сдвигаемый элемент обозначается черной стрелкой. При обнаружении подходящей позиции новый элемент вставляется в образовавшуюся свободную ячейку.

Обратите внимание: наша реализация приоритетной очереди не поддерживает циклический перенос. Вставка выполняется медленно из-за необходимости поиска подходящей позиции, зато извлечение происходит быстро. Поддержка циклического переноса положения не исправит. Также обратите внимание на то, что стрелка `Rear` не перемещается; она всегда указывает на ячейку с индексом 0 у нижней границы массива.

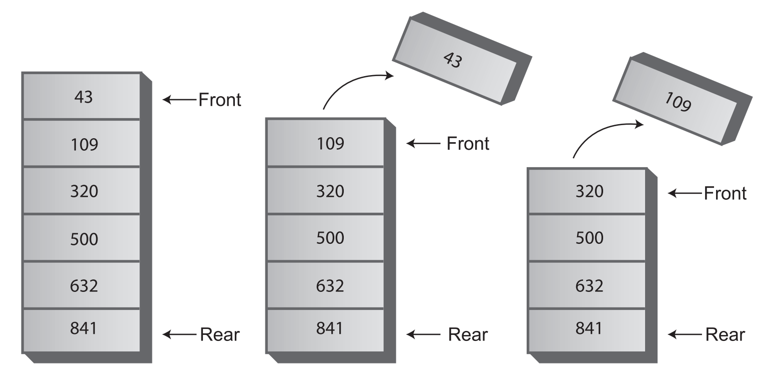
Извлечение

Извлекаемый элемент всегда находится на верхней границе массива, поэтому операция выполняется быстро и просто; элемент удаляется, а стрелка `Front` перемещается к новой вершине массива. Ни сдвиги, ни сравнения не нужны.

В приложении PriorityQ Workshop стрелки `Front` и `Rear` отображаются для сравнения с обычной очередью, но на самом деле они не нужны. Алгоритму известно, что начало очереди всегда находится на вершине массива (`nItems-1`), поэтому элементы вставляются в соответствии с приоритетами, а не «с хвоста» очереди. Схема работы методов класса `PriorityQ` показана на рис. 4.12.



Вставка нового элемента в приоритетную очередь



Извлечение двух элементов из приоритетной очереди

Рис. 4.12. Логика методов класса PriorityQ

Чтение и инициализация

Кнопка Peek читает значение элемента с наименьшим ключом (без извлечения из очереди), а кнопка New создает новую пустую приоритетную очередь.

Другие возможные реализации

Реализация, представленная в приложении PriorityQ Workshop, недостаточно эффективна — при выполнении вставки приходится перемещать в среднем половину элементов.

Другое решение (также реализованное на базе массива) не пытается сохранить порядок сортировки элементов — новые элементы просто вставляются на вершину массива. В этом случае вставка выполняется очень быстро, но, к сожалению, замедляется извлечение из-за необходимости поиска элемента с наименьшим ключом. Такой подход требует просмотра всех элементов и сдвига (в среднем) половины из них для заполнения свободного места. Как правило, подход с быстрым извлечением, продемонстрированный в приложении, является предпочтительным.

При малом количестве элементов и в ситуациях, в которых скорость не критична, достаточно реализации приоритетной очереди на базе массива. При большом количестве элементов и в ситуациях, критичных по скорости, лучше воспользоваться реализацией на базе кучи.

Реализация приоритетной очереди на языке Java

Реализация приоритетной очереди на базе простого массива на языке Java представлена в листинге 4.6.

Листинг 4.6. Программа priorityQ.java

```
// priorityQ.java
// Работа с приоритетной очередью
// Запуск программы: C>java PriorityQApp
////////////////////////////////////////////////////////////////
class PriorityQ
{
   // Элементы массива сортируются по значению ключа,
   // от максимума (0) до минимума (maxSize-1)
   private int maxSize;
   private long[] queArray;
   private int nItems;
//-------------------------------------------------------------
   public PriorityQ(int s)          // Конструктор
      {
      maxSize = s;
      queArray = new long[maxSize];
      nItems = 0;
      }
//-------------------------------------------------------------
```

```
   public void insert(long item)    // Вставка элемента
      {
      int j;

      if(nItems==0)                        // Если очередь пуста,
         queArray[nItems++] = item;        // вставляем в ячейку 0
      else                                 // Если очередь содержит элементы
         {
         for(j=nItems-1; j>=0; j--)        // Перебор в обратном направлении
            {
            if( item > queArray[j] )          // Если новый элемент больше,
               queArray[j+1] = queArray[j]; // сдвинуть вверх
            else                              // Если меньше,
               break;                         // сдвиг прекращается
            }
         queArray[j+1] = item;                // Вставка элемента
         nItems++;
         }
      }  //
//-------------------------------------------------------------
   public long remove()             // Извлечение минимального элемента
      { return queArray[--nItems]; }
//-------------------------------------------------------------
   public long peekMin()            // Чтение минимального элемента
      { return queArray[nItems-1]; }
//-------------------------------------------------------------
   public boolean isEmpty()         // true, если очередь пуста
      { return (nItems==0); }
//-------------------------------------------------------------
   public boolean isFull()          // true, если очередь заполнена
      { return (nItems == maxSize); }
//-------------------------------------------------------------
   }  // Конец класса PriorityQ
////////////////////////////////////////////////////////////////
class PriorityQApp
   {
   public static void main(String[] args) throws IOException
      {
      PriorityQ thePQ = new PriorityQ(5);
      thePQ.insert(30);
      thePQ.insert(50);
      thePQ.insert(10);
      thePQ.insert(40);
      thePQ.insert(20);

      while( !thePQ.isEmpty() )
         {
         long item = thePQ.remove();
         System.out.print(item + " ");  // 10, 20, 30, 40, 50
         }
```

```
      System.out.println("");
      }
//-------------------------------------------------------------
   }  // Конец класса PriorityQApp
```

В методе main() мы сначала вставляем в очередь пять элементов в произвольном порядке, а затем извлекаем и выводим их. Первым всегда извлекается метод с наименьшим ключом, поэтому результат выглядит так:

```
10, 20, 30, 40, 50
```

Метод insert() сначала проверяет, содержит ли очередь хотя бы один элемент; если элементы отсутствуют, элемент вставляется в ячейку с индексом 0. В противном случае он начинает с вершины массива сдвигать существующие элементы вверх, пока не найдет место, подходящее для вставки нового элемента. Затем элемент вставляется, а значение nItems увеличивается. Если существует вероятность того, что приоритетная очередь заполнена, следует проверить эту возможность методом isFull() перед вызовом insert().

Поля front и rear в этой реализации не нужны (в отличие от обычной очереди), потому что, как уже говорилось ранее, значение front всегда равно nItems-1, а значение rear всегда равно 0.

Метод remove() предельно прост: он уменьшает nItems и возвращает элемент, находящийся на вершине массива. Метод peekMin() устроен аналогично, если не считать того, что он не уменьшает nItems. Методы isEmpty() и isFull() проверяют, содержит ли поле nItems значение 0 или maxSize соответственно.

Эффективность приоритетных очередей

В рассмотренной реализации приоритетной очереди вставка выполняется за время $O(N)$, а извлечение — за время $O(1)$. В главе 12 будет показано, как улучшить время вставки за счет реализации на базе кучи.

Разбор арифметических выражений

Итак, в этой главе были описаны три разные структуры данных. Давайте немного сменим тему и рассмотрим важное практическое применение одной из этих структур.

Приложение разбирает (то есть анализирует) арифметические выражения вида 2 + 3, или 2 × (3 + 4), или ((2 + 4) × 7) + 3 × (9 – 5). Для хранения компонентов выражения используется стек. В программе brackets.java (см. листинг 4.3) был представлен пример использования стека для проверки парности скобок. Аналогичным способом (хотя и более сложным) стеки могут применяться и для разбора арифметических выражений.

Материал данного раздела в каком-то смысле можно рассматривать как «дополнительное чтение». Он не обязателен для понимания книги, а в вашей повседневной работе вам вряд ли придется часто писать код разбора арифметических выражений (если, конечно, вы не занимаетесь разработкой компиляторов). Кроме того, реализация программы намного сложнее всего, что рассматривалось нами

ранее. И все же это важное практическое применение стеков чрезвычайно поучительно, а многие его аспекты представляют интерес сами по себе.

Как оказалось, вычислить результат арифметического выражения довольно трудно (во всяком случае для компьютерного алгоритма). Проще всего разделить этот процесс на два шага:

1. Преобразование арифметического выражения в другой формат, называемый *постфиксной записью*.
2. Вычисление результата по постфиксной записи.

Шаг 1 относительно сложен, зато шаг 2 выполняется просто. В любом случае двухшаговая схема обеспечивает более простой алгоритм, чем попытка прямого разбора арифметических выражений. Конечно, человеку проще работать с обычными арифметическими выражениями. Вскоре мы вернемся к различиям в подходах человека и компьютера к решению этой задачи.

Прежде чем переходить к рассмотрению всех подробностей шагов 1 и 2, необходимо познакомиться с постфиксной записью.

Постфиксная запись

В записи традиционных арифметических выражений оператор (+, –, * или /) размещается между двумя операндами (числа или обозначающие их символические имена). Запись, в которой оператор записывается между операндами, называется *инфиксной*. Мы пишем 2 + 2, или 4 × 7, или с обозначением чисел буквами — A + B и A/B.

В *постфиксной* записи, также называемой *обратной польской записью*, или RPN (Reverse Polish Notation), потому что она была изобретена польским математиком, оператор записывается после двух операндов. Таким образом, A + B превращается в AB+, а A/B превращается в AB/. Более сложные инфиксные выражения тоже преобразуются в постфиксную запись; примеры приведены в табл. 4.2. Вскоре вы узнаете, как строятся постфиксные выражения.

Таблица 4.2. Инфиксные и постфиксные выражения

Инфиксная запись	Постфиксная запись
A+B–C	AB+C–
A*B/C	AB*C/
A+B*C	ABC*+
A*B+C	AB*C+
A*(B+C)	ABC+*
A*B+C*D	AB*CD*+
(A+B)*(C–D)	AB+CD–*
((A+B)*C)–D	AB+C*D–
A+B*(C–D/(E+F))	ABCDEF+/–*+

В некоторых языках программирования также существует оператор возведения в степень (чаще всего символ ^), но в нашем обсуждении эта возможность не рассматривается.

Кроме инфиксной и постфиксной, также существует *префиксная запись*, при которой оператор записывается перед операндами: +AB вместо AB+. Эта запись функционально эквивалентна постфиксной, но на практике она применяется редко.

Преобразование инфиксной записи в постфиксную

На нескольких ближайших страницах подробно рассматривается процесс преобразования выражений из инфиксной записи в постфиксную. Алгоритм не из простых, так что не огорчайтесь, если что-то покажется непонятным. Если вы чувствуете, что начинаете вязнуть в теории, переходите к разделу «Вычисление постфиксных выражений». Чтобы понять, как создаются постфиксные выражения, полезно рассмотреть процесс вычисления их результата — например, вычисления значения 14 для выражения 234 + ×, постфиксного эквивалента 2 × (3 + 4). (Для простоты мы ограничимся операндами, состоящими из одной цифры, хотя в выражениях также могут использоваться и операнды из нескольких цифр).

Как мы вычисляем результаты инфиксных выражений

Как преобразовать инфиксную запись в постфиксную? Для начала зададим себе чуть более простой вопрос: а как человек вычисляет результат обычного инфиксного выражения? Как говорилось ранее, компьютеру это сделать непросто, но мы легко справляемся с задачей благодаря школьным урокам математики. Любой читатель легко вычислит результат выражения 3 + 4 + 5 или 3 × (4 + 5). Анализ последовательности вычисления даст нам некоторое представление о том, как такие выражения преобразуются в постфиксную запись.

Упрощенно говоря, мы «обрабатываем» арифметические выражения по следующим правилам:

1. Выражение читается слева направо. (По крайней мере будем предполагать, что это так. Некоторые люди склонны забегать вперед, но мы будем считать, что вы читаете по порядку, слева направо.)
3. Прочитав достаточно символов, чтобы определить два операнда и оператор, вы выполняете вычисления и подставляете результат на место этих двух операндов с оператором. (Возможно, вам также придется обработать незавершенные операции слева — об этом чуть позже.)
4. Этот процесс — перемещение слева направо и вычисление результата там, где это возможно, — продолжается до конца выражения.

В таблицах 4.3–4.5 приведены примеры вычисления результата простых инфиксных выражений. Далее в табл. 4.6–4.8 видно, как близко эти вычисле-

ния воспроизводят процесс преобразования инфиксных выражений в постфиксные.

Чтобы вычислить результат выражения 3 + 4 – 5, необходимо выполнить действия из табл. 4.3.

Таблица 4.3. Вычисление результата выражения 3 + 4 – 5

Прочитанный элемент	Обработанная часть выражения	Комментарии
3	3	
3+	3+	
4	3 + 4	
–	7	При достижении знака можно вычислять результат подвыражения 3 + 4.
	7–	
5	7 – 5	
конец	2	При достижении конца выражения можно вычислять результат подвыражения 7 – 5.

Мы можем вычислить результат 3 + 4 только после того, как увидим, какой оператор следует за 4. Если это оператор * или /, то применение оператора + придется отложить до вычисления результата оператора * или /.

Однако в нашем примере за 4 следует оператор –, который обладает одинаковым приоритетом с +. Обнаружив оператор –, мы знаем, что можем вычислить результат 3 + 4 — он равен 7. Таким образом, подвыражение 3 + 4 заменяется результатом 7. Результат подвыражения 7 – 5 вычисляется при достижении конца выражения.

На рис. 4.13 этот процесс изображен более подробно. Обратите внимание: входные символы читаются слева направо, а при накоплении достаточной информации мы двигаемся справа налево, восстанавливаем ранее просмотренные входные данные и вычисляем результат каждой комбинации «операнд-оператор-операнд».

Из-за разных приоритетов обработка выражения 3 + 4 × 5 проходит более сложно. Выполняемые действия перечислены в табл. 4.4.

Таблица 4.4. Вычисление результата выражения 3 + 4 × 5

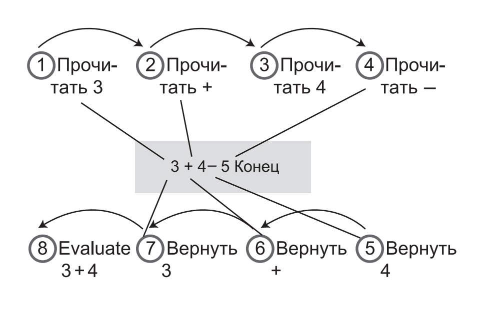
Прочитанный элемент	Обработанная часть выражения	Комментарии
3	3	
+	3 +	
4	3 + 4	
*	3 + 4 ×	Вычислять 3 + 4 еще рано, потому что оператор * обладает более высоким приоритетом, чем оператор +

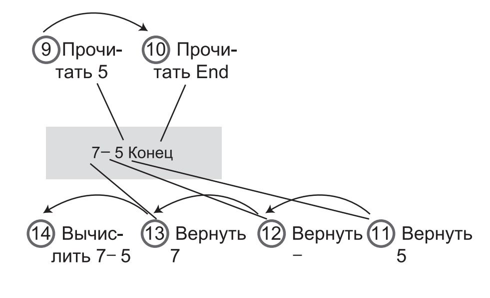
продолжение ⇗

Таблица 4.4 (*продолжение*)

Прочитанный элемент	Обработанная часть выражения	Комментарии
5	3 + 4 × 5 3 + 20	При достижении операнда 5 можно вычислять 4 × 5.
Конец	23	При достижении конца выражения можно вычислять результат подвыражения 3 + 20

Сложение 3 + 4 не может быть выполнено до вычисления результата 4 × 5. Почему? Потому что умножение обладает более высоким приоритетом, чем сложение. Более того, оба оператора * и / обладают более высоким приоритетом, чем + и —, поэтому все операции умножения и деления должны быть выполнены до операций сложения и вычитания (если только круглые скобки не изменяют приоритет операций, как в следующем примере).





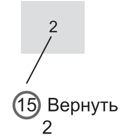


Рис. 4.13. Вычисление результата 3 + 4 – 5

Часто результат вычисляется при перемещении слева направо, как в предыдущем примере. Однако при обнаружении комбинаций «операнд-оператор-операнд»

(например, A + B) необходимо убедиться в том, что оператор справа от B не обладает более высоким приоритетом, чем + — в этом случае (как в нашем примере) выполнять сложение еще рано. После чтения 5 умножение уже возможно, потому что оно заведомо обладает самым высоким приоритетом, даже если за 5 следует * или /. С другой стороны, сложение возможно лишь после того, как вы узнаете, какой оператор следует за 5. Когда становится ясно, что за 5 нет ничего, кроме конца выражения, можно переходить к вычислению. Процесс изображен на рис. 4.14.

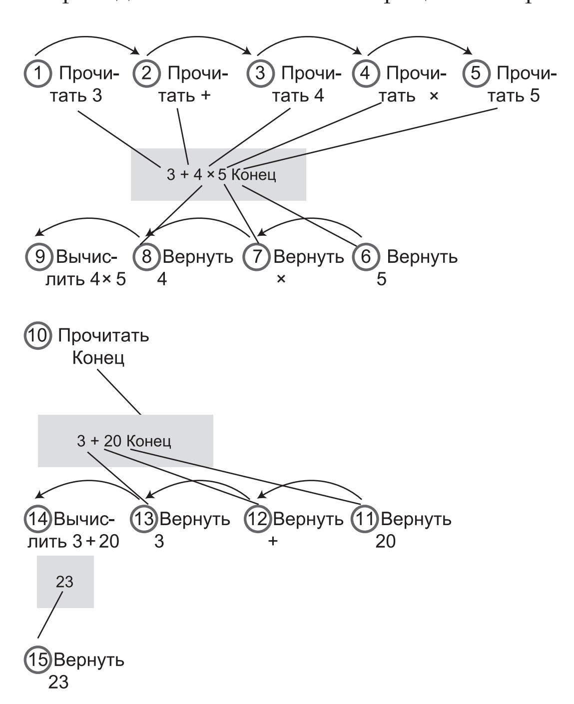


Рис. 4.14. Вычисление результата 3 + 4 × 5

Круглые скобки переопределяют стандартный приоритет операторов. В таблице 4.5 описан порядок вычисления обработки выражения 3 × (4 + 5). Без круглых скобок сначала было бы выполнено умножение, а затем сложение.

В этом примере вычисления невозможны до того момента, когда во входном потоке встретится закрывающая скобка. Умножение обладает более высоким или равным приоритетом по сравнению с другими операторами, поэтому обычно результат 3 × 4 вычисляется сразу же после чтения 4. Однако круглые скобки по приоритету превосходят даже * и /, поэтому результат выражения в круглых скобках необходимо вычислить до использования его в качестве операнда любых других вычислений. Закрывающая скобка означает, что обработка выражения может перейти к сложению. Суммирование 4 + 5 дает 9; зная это, мы вычисляем 3 × 9 и получаем 27. Достижение конца выражения указывает на то, что символов для обработки больше нет. Процесс показан на рис. 4.15.

Таблица 4.5. Вычисление результата выражения 3 × (4 + 5)

Прочитанный элемент	Обработанная часть выражения	Комментарии
3	3	
*	3 ×	
(	3 × (	
4	3 × (4	Вычислять 3 × 4 еще рано из-за скобки
+	3 × (4 +	
5	3 × (4 + 5	Вычислять 4 + 5 еще рано.
)	3 × (4 + 5) 3 × 9 27	После) можно вычислить 4 + 5. После вычисления 4 + 5 можно вычислять 3 × 9.
Конец		Вычисления завершены

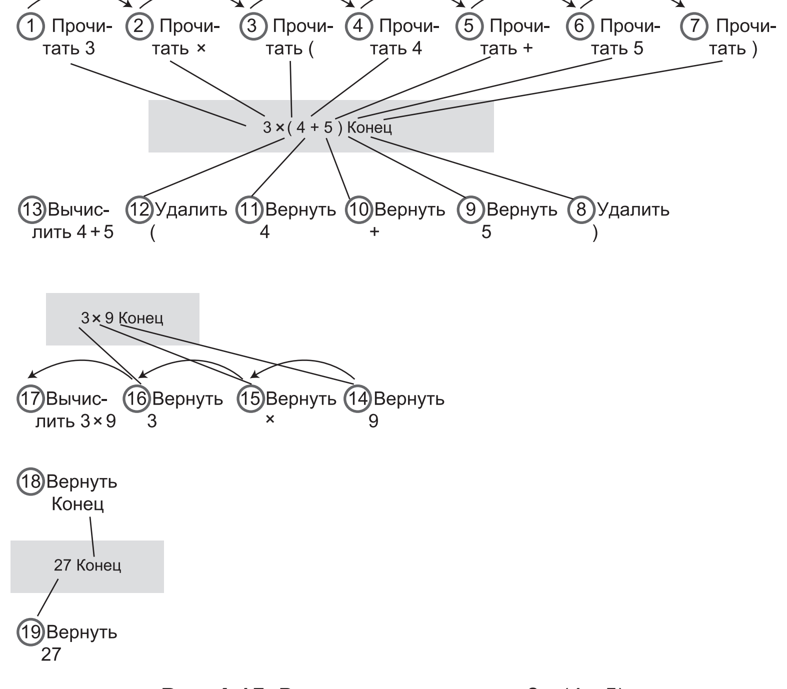


Рис. 4.15. Вычисление результата 3 × (4 + 5)

Как видите, в процессе вычисления результата инфиксного арифметического выражения приходится перемещаться по выражению в прямом и обратном направлениях. Сначала слева направо читаются операнды и операторы. При накоплении

информации, достаточной для применения оператора, вы начинаете двигаться в обратном направлении, «вспоминаете» два операнда и оператор и выполняете арифметическое действие. Если за оператором следует другой оператор с более высоким приоритетом или скобки, применение оператора откладывается. В этом случае сначала применяется следующий, более приоритетный оператор, а затем вы возвращаетесь обратно (влево) и применяете более ранние операторы.

Конечно, алгоритм для выполнения подобной обработки можно было бы запрограммировать напрямую, и все же, как уже говорилось ранее, проще сначала преобразовать выражение к постфиксной записи.

Как мы преобразуем инфиксную запись в постфиксную

Преобразование инфиксной записи в постфиксную осуществляется примерно по тем же правилам, что и вычисление результата инфиксного выражения. Тем не менее процесс содержит ряд изменений; никакие арифметические операции в нем не выполняются. Конечной целью является не вычисление результата, а расположение операторов и операндов в новом формате постфиксной записи. Результат полученного постфиксного выражения будет вычислен позднее.

Как и прежде, инфиксное выражение читается слева направо, с последовательным просмотром всех символов. В процессе чтения операторы и операнды копируются в постфиксную выходную строку. Фокус в том, чтобы понять, когда и что следует копировать.

Если символ инфиксной строки определяет операнд, он немедленно копируется в постфиксную строку. Иначе говоря, если в инфиксном выражении встречается символ A, он записывается в постфиксное выражение. Операнды всегда копируются без задержки, сразу же после их обнаружения, сколько бы ни приходилось ждать копирования связанных с ними операторов.

Выбрать момент копирования оператора сложнее, но общая процедура мало чем отличается от процедуры вычисления результата инфиксных выражений. Каждый раз, когда оператор может использоваться для вычисления результата инфиксного выражения (если бы вместо преобразования происходило вычисление), он копируется в постфиксную строку.

В таблице 4.6 представлена последовательность преобразования выражения A + B – C в постфиксную запись.

Обратите внимание на сходство этой таблицы с табл. 4.3 с описанием вычисления результата инфиксного выражения 3 + 4 – 5. В каждой точке, где в табл. 4.3 производилось вычисление, оператор просто записывается в постфиксный вывод.

В табл. 4.7 описан процесс преобразования выражения A + B × C в постфиксную форму. В целом он аналогичен вычислению результата 3 + 4 × 5 в табл. 4.4.

Последний пример из таблицы 4.8 демонстрирует преобразование выражения A × (B + C) в постфиксную форму. Процесс аналогичен вычислению результата 3 × (4 + 5) в табл. 4.5. Постфиксные операторы могут копироваться в выходную строку только после достижения закрывающих круглых скобок во входных данных.

Таблица 4.6. Преобразование выражения A + B – C в постфиксную запись

Символ, прочитанный из инфиксного выражения	Текущее состояние обработки инфиксного выражения	Текущее состояние записи постфиксного выражения	Комментарии
A	A	A	
+	A+	A	
B	A + B	AB	
–	A + B –	AB +	При обнаружении оператора – оператор + копируется в выходную строку.
C	A + B – C	AB + C	
конец	A + B – C	AB + C –	При достижении конца выражения копируется оператор –.

Таблица 4.7. Преобразование A + B × C в постфиксную форму

Символ, прочитанный из инфиксного выражения	Текущее состояние обработки инфиксного выражения	Текущее состояние записи постфиксного выражения	Комментарии
A	A	A	
+	A +	A	
B	A + B	AB	
×	A + B ×	AB	Копировать + еще рано, потому что оператор × обладает более высоким приоритетом, чем оператор +
C	A + B × C A + B × C	ABC ABC ×	При достижении операнда C можно копировать ×
Конец	A + B × C	ABC × +	При достижении конца выражения можно копировать +

Таблица 4.8. Преобразование A × (B × C) в постфиксную форму

Символ, прочитанный из инфиксного выражения	Текущее состояние обработки инфиксного выражения	Текущее состояние записи постфиксного выражения	Комментарии
A	A	A	
×	A ×	A	
(	A × (	A	
B	A × (B	AB	Копировать × еще рано из-за скобки

Символ, прочитанный из инфиксного выражения	Текущее состояние обработки инфиксного выражения	Текущее состояние записи постфиксного выражения	Комментарии
+	A × (B +	AB	
C	A × (B + C	ABC	Копировать + еще рано
)	A × (B + C) A × (B + C)	ABC + ABC + ×	После) можно копировать + После копирования + можно скопировать ×
Конец	A × (B + C)	ABC + ×	Копирование завершено

Как и в процессе вычисления результата выражения, мы перемещаемся по инфиксному выражению в обоих направлениях. Если за оператором следует оператор с более высоким приоритетом или открывающая круглая скобка, он не может быть сразу скопирован в выходную (постфиксную) строку — в таких ситуациях высокоприоритетный оператор или оператор в скобках должен быть записан в постфиксное выражение до низкоприоритетного оператора.

Сохранение операторов в стеке

И в табл. 4.7, и в табл. 4.8 переход от инфиксной записи к постфиксной изменяет порядок следования операторов. Так как первый оператор может быть скопирован в выходной поток только после копирования второго, операторы выводятся в постфиксную строку в порядке, противоположном порядку их чтения из инфиксной строки. Возможно, более сложный пример поможет лучше понять суть происходящего. В табл. 4.9 продемонстрировано преобразование инфиксного выражения A + B × (C – D) в постфиксную форму. Смысл нового столбца («Содержимое стека») будет объяснен ниже.

Таблица 4.9. Преобразование A + B × (C – D) в постфиксную форму

Символ, прочитанный из инфиксного выражения	Текущее состояние обработки инфиксного выражения	Текущее состояние записи постфиксного выражения	Содержимое стека
A	A	A	
+	A +	A	+
B	A + B	AB	+
×	A + B ×	AB	+ ×
(	A + B × (	AB	+ × (
C	A + B × (C	ABC	+ × (
–	A + B × (C –	ABC	+ × (–
D	A + B × (C – D	ABCD	+ × (–

продолжение ⇗

Таблица 4.9 (*продолжение*)

Символ, прочитанный из инфиксного выражения	Текущее состояние обработки инфиксного выражения	Текущее состояние записи постфиксного выражения	Содержимое стека
)	A + B × (C – D)	ABCD –	+ × (
	A + B × (C – D)	ABCD –	+ × (
	A + B × (C – D)	ABCD –	+ ×
	A + B × (C – D)	ABCD – ×	+
	A + B × (C – D)	ABCD – × +	

Мы видим, что в исходном инфиксном выражении операторы следуют в порядке + × –, а в итоговом постфиксном выражении используется обратный порядок – × +. Это объясняется тем, что оператор × имеет более высокий приоритет, чем +, а оператор –, заключенный в круглые скобки, по приоритету превосходит ×. Перестановка в обратном порядке наводит на мысль о том, что для хранения операторов в ожидании их использования хорошо подойдет стек. Последний столбец в табл. 4.9 показывает содержимое стека на различных стадиях процесса преобразования.

Извлечение элементов из стека позволяет вам в некотором смысле перемещаться в обратном направлении (справа налево) по входной строке. На самом деле анализируется не вся входная строка, а только входящие в нее операторы и круглые скобки. Они заносятся в стек в процессе чтения входной строки, поэтому для перестановки их в обратном порядке достаточно последовательно извлекать их из стека. Порядок следования операндов (A, B и т. д.) не отличается в постфиксной и инфиксной записи, поэтому операнды можно просто записывать в выходной поток сразу же после обнаружения; сохранять их в стеке не обязательно.

Правила преобразования

Давайте сформулируем более конкретные правила преобразования инфиксной записи в постфиксную. С элементами, последовательно читаемыми из инфиксной входной строки, выполняются действия из табл. 4.10. Действия описаны на псевдокоде.

Таблица 4.10. Правила преобразования инфиксной записи в постфиксную

Элемент, прочитанный из входной (инфиксной) строки	Действие
Операнд	Записать в выходную (постфиксную) строку
Открывающая круглая скобка (	Занести в стек
Закрывающая круглая скобка)	Пока в стеке остаются элементы: – Извлечь элемент, – Если элемент отличен от (, записать в выходную строку. При чтении элемента) выйти из строки

Элемент, прочитанный из входной (инфиксной) строки	Действие
Оператор (opThis)	Если стек пуст, – Занести opThis в стек Иначе – Пока в стеке остаются элементы: –– Извлечь элемент, –– Если это элемент (, занести его в стек, или –– Если это оператор (opTop), и ––– Если opTop < opThis, занести в стек opTop, или ––– Если opTop >= opThis, скопировать opTop в выходную строку. – Выйти из цикла, если opTop < opThis, или при извлечении (. – Занести в стек opThis
Конец входных данных	Пока в стеке остаются элементы, – Извлечь элемент, скопировать в выходную строку

В этой таблице обозначения < и >= относятся к приоритетам операторов, а не к числовым значениям. Оператор `opThis` только что был прочитан из инфиксной входной строки, а оператор `opTop` только что был извлечен из стека.

Убедитесь в том, что этот набор правил действительно работает. В табл. 4.11–4.13 показано, как эти правила применяются к трем инфиксным выражениям. Эти таблицы аналогичны табл. 4.6–4.8, не считая того, что для каждого шага добавлены соответствующие правила. Попробуйте построить такие же таблицы для других инфиксных выражений.

Таблица 4.11. Применение правил преобразования к A + B – C

Символ, прочитанный из инфиксного выражения	Текущее состояние обработки инфиксного выражения	Текущее состояние записи постфиксного выражения	Содержимое стека	Правило
A	A	A		Записать операнд в выходную строку
+	A +	A	+	Если стек пуст, занести opThis
B	A + B	AB	+	Записать операнд в выходную строку
–	A + B – A + B – A + B –	AB AB + AB +	 –	Стек не пуст, извлечь элемент opTop (+) >= opThis (–), записать opTop Занести в стек opThis
C	A + B – C	AB + C	–	Записать операнд в выходную строку
Конец	A + B – C	AB + C –		Извлечь последний элемент, записать в выходную строку

Таблица 4.12. Применение правил преобразования к A + B × C

Символ, прочитанный из инфиксного выражения	Текущее состояние обработки инфиксного выражения	Текущее состояние записи постфиксного выражения	Содержимое стека	Правило
A	A	A		Записать операнд в выходную строку.
+	A +	A	+	Если стек пуст, занести opThis.
B	A + B	AB	+	Записать операнд в выходную строку
×	A + B × A + B × A + B ×	AB AB AB	+ + + ×	Стек не пуст, извлечь opTop. opTop (+) < opThis (×), занести в стек opTop. Занести в стек opThis.
C	A + B × C	ABC	+ ×	Записать операнд в выходную строку.
Конец	A + B × C A + B × C	ABC × ABC × +	+	Извлечь элемент, записать в выходную строку. Извлечь элемент, записать в выходную строку

Таблица 4.13. Применение правил преобразования к A × (B + C)

Символ, прочитанный из инфиксного выражения	Текущее состояние обработки инфиксного выражения	Текущее состояние записи постфиксного выражения	Содержимое стека	Правило
A	A	A		Записать операнд в выходную строку
×	A ×	A	×	Если стек пуст, занести в стек opThis
(	A × (	A	× (	Занести (в стек
B	A × (B	AB	× (	Записать операнд в выходную строку
+	A × (B + A × (B + A × (B +	AB AB AB	× × (× (+	Стек не пуст, извлечь элемент. Это (, занести в стек. Затем занести в стек opThis
C	A × (B + C	ABC	× (+	Записать операнд в выходную строку
)	A × (B + C) A × (B + C)	ABC + ABC +	× (×	Извлечь элемент, записать в выходную строку. Прекратить извлечение, если извлечен элемент (
Конец	A × (B + C)	ABC + ×		Извлечь оставшийся элемент, записать в выходную строку

Преобразование инфиксной записи в постфиксную на языке Java

В листинге 4.7 приведена программа `infix.java`, использующая правила из табл. 4.10 для преобразования инфиксного выражения в постфиксное.

Листинг 4.7. Программа infix.java

```
// infix.java
// Преобразование инфиксных арифметических выражений в постфиксные
// Запуск программы: C>java InfixApp
import java.io.*;            // Для ввода/вывода
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;
   private char[] stackArray;
   private int top;
//--------------------------------------------------------------
   public StackX(int s)       // Конструктор
      {
      maxSize = s;
      stackArray = new char[maxSize];
      top = -1;
      }
//--------------------------------------------------------------
   public void push(char j)  // Размещение элемента на вершине стека
      { stackArray[++top] = j; }
//--------------------------------------------------------------
   public char pop()         // Извлечение элемента с вершины стека
      { return stackArray[top--]; }
//--------------------------------------------------------------
   public char peek()        // Чтение элемента с вершины стека
      { return stackArray[top]; }
//--------------------------------------------------------------
   public boolean isEmpty()  // true, если стек пуст
      { return (top == -1); }
//-------------------------------------------------------------
   public int size()         // Текущий размер стека
      { return top+1; }
//--------------------------------------------------------------
   public char peekN(int n)  // Чтение элемента с индексом n
      { return stackArray[n]; }
//--------------------------------------------------------------
   public void displayStack(String s)
      {
      System.out.print(s);
      System.out.print("Stack (bottom-->top): ");
      for(int j=0; j<size(); j++)
```

продолжение ➪

Листинг 4.7 (*продолжение*)

```
         {
         System.out.print( peekN(j) );
         System.out.print(' ');
         }
      System.out.println("");
      }
//--------------------------------------------------------------
   }  // Конец класса StackX
////////////////////////////////////////////////////////////////
class InToPost     // Преобразование инфиксной записи в постфиксную
   {
   private StackX theStack;
   private String input;
   private String output = "";
//--------------------------------------------------------------
   public InToPost(String in)   // Конструктор
      {
      input = in;
      int stackSize = input.length();
      theStack = new StackX(stackSize);
      }
//--------------------------------------------------------------
   public String doTrans()      // Преобразование в постфиксную форму
      {
      for(int j=0; j<input.length(); j++)
         {
         char ch = input.charAt(j);
         theStack.displayStack("For "+ch+" "); // *диагностика*
         switch(ch)
            {
            case '+':               // + или -
            case '-':
               gotOper(ch, 1);      // Извлечение операторов
               break;               //   (приоритет 1)
            case '*':               // * или /
            case '/':
               gotOper(ch, 2);      // Извлечение операторов
               break;               //   (приоритет 2)
            case '(':               // Открывающая круглая скобка
               theStack.push(ch);   // Занести в стек
               break;
            case ')':               // Закрывающая круглая скобка
               gotParen(ch);        // Извлечение операторов
               break;
            default:                // Остается операнд
               output = output + ch; // Записать в выходную строку
               break;
            }
         }
```

```
      while( !theStack.isEmpty() )     // Извлечение оставшихся операторов
         {
         theStack.displayStack("While ");  // *диагностика*
         output = output + theStack.pop(); // write to output
         }
      theStack.displayStack("End   ");     // *диагностика*
      return output;                   // Возвращение постфиксного выражения
      }
//--------------------------------------------------------------
   public  void gotOper(char opThis, int prec1)
      {                                // Чтение оператора из входной строки
      while( !theStack.isEmpty() )
         {
         char opTop = theStack.pop();
         if( opTop == '(' )            // Если это '('
            {
            theStack.push(opTop);      // Вернуть '('
            break;
            }
         else                          // Оператор
            {
               int prec2;                   // Приоритет нового оператора
               if(opTop=='+' || opTop=='-') // Определение приоритета
                  prec2 = 1;
               else
                  prec2 = 2;
               if(prec2 < prec1)          // Если приоритет нового оператора
                  {                       // меньше приоритета старого
                  theStack.push(opTop);   // Сохранить новый оператор
                  break;
                  }
               else                          // Приоритет нового оператора
                  output = output + opTop;  // не меньше приоритета старого
               }
            }
         theStack.push(opThis);        // Занесение в стек нового оператора
         }
//--------------------------------------------------------------
   public  void gotParen(char ch)
      {                             // Прочитана закрывающая скобка
      while( !theStack.isEmpty() )
         {
         char chx = theStack.pop();
         if( chx == '(' )           // Если извлечен элемент '('
            break;                  // Прервать выполнение
         else                       // Если извлечен оператор
            output = output + chx;  // Вывести в постфиксную строку
         }
      }
```

продолжение ⇨

Листинг 4.7 (*продолжение*)

```
//--------------------------------------------------------------
   }  // Конец класса InToPost
////////////////////////////////////////////////////////////////
class InfixApp
   {
   public static void main(String[] args) throws IOException
      {
      String input, output;
      while(true)
         {
         System.out.print("Enter infix: ");
         System.out.flush();
         input = getString();         // Чтение строки с клавиатуры
         if( input.equals("") )       // Выход, если нажата клавиша [Enter]
            break;
                                      // Создание объекта-преобразователя
         InToPost theTrans = new InToPost(input);
         output = theTrans.doTrans(); // Преобразование
         System.out.println("Postfix is " + output + '\n');
         }
      }
//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   }  // Конец класса InfixApp
////////////////////////////////////////////////////////////////
```

Метод main() класса InfixApp запрашивает у пользователя инфиксное выражение. Для ввода данных используется вспомогательный метод readString(). Программа создает объект InToPost, инициализируемый входной строкой, после чего вызывает метод doTrans() для выполнения преобразования. Метод возвращает постфиксную выходную строку, которая выводится на экран.

Правила преобразования из табл. 4.10 реализуются конструкцией switch метода doTrans(). При чтении оператора вызывается метод gotOper(), а при чтении закрывающей круглой скобки) — метод gotParen(). Эти методы реализуют третье и четвертое правило из таблицы, более сложные по сравнению с другими правилами.

В класс StackX включен метод displayStack() для вывода всего содержимого стека. Теоретически это неправильно; предполагается, что при работе со стеком доступен только элемент, находящийся на вершине. Однако этот метод является

удобным диагностическим средством для просмотра содержимого стека на каждой стадии преобразования. Пример выполнения программы infix.java:

```
Enter infix: A*(B+C)-D/(E+F)
For A Stack (bottom-->top):
For * Stack (bottom-->top):
For ( Stack (bottom-->top): *
For B Stack (bottom-->top): * (
For + Stack (bottom-->top): * (
For C Stack (bottom-->top): * ( +
For ) Stack (bottom-->top): * ( +
For - Stack (bottom-->top): *
For D Stack (bottom-->top): -
For / Stack (bottom-->top): -
For ( Stack (bottom-->top): - /
For E Stack (bottom-->top): - / (
For + Stack (bottom-->top): - / (
For F Stack (bottom-->top): - / ( +
For ) Stack (bottom-->top): - / ( +
While Stack (bottom-->top): - /
While Stack (bottom-->top): -
End   Stack (bottom-->top):
Postfix is ABC+*DEF+/-
```

Выходные данные показывают, где вызывается метод displayStack() (из цикла for, из цикла while или в конце программы), а для цикла for — какой символ был только что прочитан из входной строки.

Вместо символических имен типа A или B можно использовать цифры (например, 3 или 7) — программа тоже интерпретирует их как символы. Пример:

```
Enter infix: 2+3*4
For 2 Stack (bottom-->top):
For + Stack (bottom-->top):
For 3 Stack (bottom-->top): +
For * Stack (bottom-->top): +
For 4 Stack (bottom-->top): + *
While Stack (bottom-->top): + *
While Stack (bottom-->top): +
End   Stack (bottom-->top):
Postfix is 234*+
```

Разумеется, в постфиксной записи 234 означает три разных операнда 2, 3 и 4.

Программа infix.java не проверяет входные данные на наличие ошибок. Если ввести некорректное инфиксное выражение, программа выдаст некорректный результат, или попросту «упадет». Поэкспериментируйте с программой. Начните с простых инфиксных выражений, попробуйте предсказать вид постфиксного выражения, затем запустите программу и проверьте ответ. Довольно скоро вы станете экспертом в области постфиксных выражений.

Вычисление результата постфиксного выражения

Как видите, преобразование инфиксного выражения в постфиксное — не самая тривиальная задача. Так ли уж необходимы все эти хлопоты? Да, преимущества новой записи проявляются при вычислении результата постфиксных выражений. Но прежде чем показывать, насколько упрощается алгоритм, давайте посмотрим, как подобные вычисления выполняются человеком.

Как мы вычисляем результат постфиксного выражения

На рис. 4.16 показано, как человек вычисляет результат постфиксного выражения, пользуясь карандашом и собственными глазами.

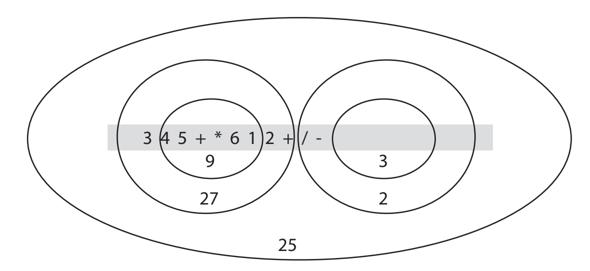


Рис. 4.16. Визуальное вычисление результата постфиксного выражения 345+*612+/-

Начните с первого оператора от левого края выражения. Обведите кружком его и два операнда слева от него. Примените оператор к этим двум операндам, выполните соответствующие вычисления и запишите результат в кружке. На рисунке вычисление 4 + 5 дает 9.

Теперь перейдите к следующему оператору справа и обведите кружком его, уже нарисованный кружок и операнд слева от него. Примените оператор к предыдущему результату и новому операнду, запишите результат в новом кружке ($3 \times 9 = 27$). Продолжайте, пока не будут применены все операторы: 1 + 2 = 3, 6/3 = 2. Ответом является результат в самом большом кружке: 27 – 2 = 25.

Правила вычисления результатов постфиксных выражений

Как написать программу, реализующую этот процесс вычисления? Как видно из предшествующего описания, оператор каждый раз применяется к двум последним операндам, которые вы встречаете при обработке выражения. Это наводит на мысль, что операнды будет уместно хранить в стеке (в отличие от алгоритма преобразования инфиксной записи в постфиксную, в котором в стеке сохранялись *операторы*). Правила вычисления результата постфиксных выражений перечислены в табл. 4.14.

Когда обработка будет завершена, остается только извлечь окончательный результат из стека. Вот и все! Этот процесс является компьютерным аналогом процесса с рисованием кружков на рис. 4.16.

Таблица 4.14. Вычисление результата постфиксного выражения

Символ, прочитанный из инфиксного выражения	Действие
Операнд	Занести в стек
Оператор	Извлечь два операнда из стека и применить к ним оператор. Занести результат в стек

Вычисление результата постфиксных выражений на языке Java

В преобразовании инфиксных выражений в постфиксные числа обозначались символами (A, B и т. д.). Такое решение работало, потому что мы не выполняли с операндами арифметические операции, а только записывали их в другом формате.

Теперь мы собираемся вычислить результат постфиксного выражения, а это потребует выполнения арифметических операций и получения ответа. Следовательно, операнды должны быть настоящими числами. Для упрощения программирования будем считать, что операнды состоят только из одной цифры.

Наша программа вычисляет и выводит результат постфиксного выражения. Не забывайте, что операнды содержат не более одной цифры. Пример простого взаимодействия с программой:

```
Enter postfix: 57+
5 Stack (bottom-->top):
7 Stack (bottom-->top): 5
+ Stack (bottom-->top): 5 7
Evaluates to 12
```

Входная строка содержит только цифры и операторы без пробелов. Программа находит числовой эквивалент выражения. Хотя операнды могут состоять только из одной цифры, на результаты (в том числе и промежуточные) это ограничение не распространяется. Как и в программе `infix.java`, содержимое стека на каждом шаге выводится методом `displayStack()`. Код программы `postfix.java` приведен в листинге 4.8.

Листинг 4.8. Программа postfix.java

```
// postfix.java
// Разбор постфиксных арифметических выражений
// Запуск программы: C>java PostfixApp
import java.io.*;              // Для ввода/вывода
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;
   private int[] stackArray;
   private int top;
```

продолжение ⇨

Листинг 4.8 (*продолжение*)

```
//--------------------------------------------------------------
   public StackX(int size)      // Конструктор
      {
      maxSize = size;
      stackArray = new int[maxSize];
      top = -1;
      }
//--------------------------------------------------------------
   public void push(int j)     // Размещение элемента на вершине стека
      { stackArray[++top] = j; }
//--------------------------------------------------------------
   public int pop()            // Извлечение элемента с вершины стека
      { return stackArray[top--]; }
//--------------------------------------------------------------
   public int peek()           // Чтение элемента на вершине стека
      { return stackArray[top]; }
//--------------------------------------------------------------
   public boolean isEmpty()    // true, если стек пуст
      { return (top == -1); }
//--------------------------------------------------------------
   public boolean isFull()     // true, если стек заполнен
      { return (top == maxSize-1); }
//--------------------------------------------------------------
   public int size()           // Текущий размер стека
      { return top+1; }
//--------------------------------------------------------------
   public int peekN(int n)     // Чтение элемента с индексом n
      { return stackArray[n]; }
//--------------------------------------------------------------
   public void displayStack(String s)
      {
      System.out.print(s);
      System.out.print("Stack (bottom-->top): ");
      for(int j=0; j<size(); j++)
         {
         System.out.print( peekN(j) );
         System.out.print(' ');
         }
      System.out.println("");
      }
//--------------------------------------------------------------
   }  // Конец класса StackX
////////////////////////////////////////////////////////////////
class ParsePost
   {
   private StackX theStack;
   private String input;
//--------------------------------------------------------------
   public ParsePost(String s)
```

```
      { input = s; }
//--------------------------------------------------------------
   public int doParse()
      {
      theStack = new StackX(20);             // Создание объекта стека
      char ch;
      int j;
      int num1, num2, interAns;

      for(j=0; j<input.length(); j++)       // Для каждого символа
         {
         ch = input.charAt(j);              // Чтение символа
         theStack.displayStack(""+ch+" ");  // *диагностика*
         if(ch >= '0' && ch <= '9')         // Если это цифра
            theStack.push( (int)(ch-'0') ); // Занести в стек
         else                               // Если это оператор
            {
            num2 = theStack.pop();          // Извлечение операндов
            num1 = theStack.pop();
            switch(ch)                      // Выполнение арифметической
               {                            // операции
               case '+':
                  interAns = num1 + num2;
                  break;
               case '-':
                  interAns = num1 - num2;
                  break;
               case '*':
                  interAns = num1 * num2;
                  break;
               case '/':
                  interAns = num1 / num2;
                  break;
               default:
                  interAns = 0;
               }
            theStack.push(interAns);        // Занесение промежуточного
            }                               // результата в стек
         }
      interAns = theStack.pop();            // Получение результата
      return interAns;
      }
   }  // Конец класса ParsePost
////////////////////////////////////////////////////////////////
class PostfixApp
   {
   public static void main(String[] args) throws IOException
      {
      String input;
```

продолжение ⇗

Листинг 4.8 (*продолжение*)

```
      int output;

      while(true)
         {
         System.out.print("Enter postfix: ");
         System.out.flush();
         input = getString();         // Ввод строки с клавиатуры
         if( input.equals("") )       // Завершение, если нажата клавиша
            break;                    // [Enter]
         // Создание объекта для разбора выражения
         ParsePost aParser = new ParsePost(input);
         output = aParser.doParse();  // Обработка выражения
         System.out.println("Evaluates to " + output);
         }
      }
//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   }  // Конец класса PostfixApp
////////////////////////////////////////////////////////////////
```

Метод main() класса PostfixApp получает постфиксную строку от пользователя и создает объект ParsePost, инициализированный этой строкой. Затем он вызывает метод doParse() класса ParsePost для выполнения непосредственной обработки.

Метод doParse() последовательно читает символы входной строки. Если символ является цифрой, он заносится в стек; если он представляет оператор, то немедленно применяется к двум операторам на вершине стека. (Операторы заведомо находятся в стеке, потому что входная строка записана в постфиксном формате.)

Результат арифметической операции заносится в стек. После чтения и применения последнего символа (которым должен быть оператор) стек содержит только один элемент — результат вычисления всего выражения.

Пример выполнения программы для более сложной входной строки: постфиксного выражения 345 + × 612+/–, обработка которого была продемонстрирована на рис. 4.16. Это выражение соответствует инфиксной записи 3 × (4 + 5) – 6/(1 + 2). (Эквивалентное преобразование с символьными именами вместо цифр приводилось в предыдущем разделе: выражение A × (B + C) – D/(E + F) в инфиксной записи соответствует ABC + × DEF +/– в постфиксной записи.) А вот как происходит вычисление в программе postfix.java:

```
Enter postfix: 345+*612+/-
3 Stack (bottom-->top):
4 Stack (bottom-->top): 3
```

```
5 Stack (bottom-->top): 3 4
+ Stack (bottom-->top): 3 4 5
* Stack (bottom-->top): 3 9
6 Stack (bottom-->top): 27
1 Stack (bottom-->top): 27 6
2 Stack (bottom-->top): 27 6 1
+ Stack (bottom-->top): 27 6 1 2
/ Stack (bottom-->top): 27 6 3
- Stack (bottom-->top): 27 2
Evaluates to 25
```

Программа `postfix.java`, как и `infix.java` (см. листинг 4.7), не проверяет входные данные. Если ввести некорректное постфиксное выражение, то результат выполнения будет непредсказуемым.

Поэкспериментируйте с программой. Попробуйте вводить разные постфиксные выражения и понаблюдайте за их обработкой — так вы поймете суть процесса быстрее, чем если будете читать о нем.

Итоги

- Стеки, очереди и приоритетные очереди обычно используются для упрощения некоторых операций из области программирования.
- В этих структурах данных доступен только один элемент.
- В стеке доступен элемент, который был вставлен последним.
- Основные операции со стеком — вставка (занесение) элемента на вершину стека и извлечение элемента, находящегося на вершине.
- В очереди доступен элемент, который был вставлен первым.
- Основные операции с очередью — вставка элемента в конец очереди и извлечение элемента в начале очереди.
- Очередь может быть реализована в форме циклической очереди (на базе массива), в которой осуществляется циклический переход индексов от конца массива к началу.
- В приоритетной очереди доступен элемент с наименьшим (или иногда с наибольшим) значением ключа.
- Основные операции с приоритетной очередью — вставка элемента в порядке сортировки и удаление элемента с наименьшим значением ключа.
- Эти структуры данных реализуются на базе массивов или других структур (например, связанных списков).
- Обычные арифметические выражения записываются в инфиксной записи, в которой оператор располагается между двумя операндами.
- В постфиксной записи оператор записывается после двух операндов.

- ✦ Для вычисления результата арифметическое выражение обычно сначала преобразуется в постфиксную запись.
- ✦ Стек является полезным инструментом как для преобразования инфиксного выражения в постфиксное, так и для вычисления результата постфиксного выражения.

Вопросы

Следующие вопросы помогут читателю проверить качество усвоения материала. Ответы на них приведены в приложении В.

1. В стек заносятся числа 10, 20, 30 и 40, после чего из стека извлекаются три элемента. Какое число останется в стеке?
2. Какое из следующих выражений истинно?
   a) Операция извлечения из стека значительно проще, чем операция извлечения из очереди.
   b) В очереди возможен циклический перенос, а в стеке — нет.
   c) Вершина стека является полным аналогом начала очереди.
   d) Как в стеке, так и в очереди последовательно удаляемые элементы располагаются в ячейках с нарастающими значениями индексов массива.
3. Что означают термины LIFO и FIFO?
4. Стек или очередь часто используются в качестве базового механизма для реализации массивов (Да/Нет).
5. Допустим, элементы массива пронумерованы слева направо, начиная с индекса 0. На базе массива реализуется очередь людей, стоящих за билетами в кинотеатр. Первому человеку, входящему в очередь, присваивается номер 1, а окно кассира находится справа. В этой ситуации:
   a) не существует числового соответствия между индексами ячеек и номерами людей в очереди;
   b) индексы ячеек массива и номера людей в очереди возрастают в противоположных направлениях;
   c) индексы ячеек массива совпадают с номерами людей в очереди;
   d) номера людей в очереди и индексы элементов массива и индексы ячеек массива двигаются в одном направлении.
6. Как в процессе вставки и удаления элементов конкретный элемент очереди перемещается в базовом массиве: от младших индексов к старшим или от старших к младшим?
7. В очередь заносятся числа 15, 25, 35 и 45, после чего из очереди извлекаются три элемента. Какое число останется в очереди?

8. Вставка/извлечение элементов в стеке и вставка/извлечение элементов в очереди выполняются за время $O(N)$.
9. Очередь может использоваться для хранения:
   a) элементов, автоматически сортируемых при вставке;
   b) сообщений о неминуемых внешних угрозах для звездолета «Энтерпрайз»;
   c) клавиш, нажатых пользователем в процессе написания письма;
   d) символических имен обрабатываемого алгебраического выражения.
10. За какое время выполняется вставка элемента в типичной приоритетной очереди (в O-синтаксисе)?
11. Определение «приоритетная» в названии приоритетной очереди означает, что:
    a) элементы с наибольшим приоритетом вставляются первыми;
    b) программист должен организовать приоритетный доступ к базовому массиву;
    c) базовый массив сортируется в соответствии с приоритетом элементов;
    d) низкоприоритетные элементы удаляются первыми.
12. По крайней мере в одном из методов программы `priorityQ.java` (листинг 4.6) используется линейный поиск (Да/Нет).
13. Одно из различий между приоритетной очередью и упорядоченным массивом:
    a) элемент с минимальным приоритетом извлекается из массива сложнее, чем из приоритетной очереди;
    b) массив должен быть упорядочен, а для приоритетной очереди это не обязательно;
    c) элемент с максимальным приоритетом извлекается из приоритетной очереди сложнее, чем из массива;
    d) все вышеперечисленное.
14. Допустим, вы реализовали класс приоритетной очереди на базе класса `OrdArray` из программы `orderedArray.java` (листинг 2.4 в главе 2, «Массивы»). Тем самым вы получите доступ к функциональности двоичного поиска. Придется ли вам вносить изменения в класс `OrdArray` для обеспечения оптимального быстродействия приоритетной очереди?
15. Приоритетную очередь удобно использовать для хранения:
    a) пассажиров, которых такси должно подобрать в разных частях города;
    b) нажатий клавиш на клавиатуре;
    c) полей шахматной доски в игровой программе;
    d) планет в модели Солнечной системы.

Упражнения

Упражнения помогут вам глубже усвоить материал этой главы. Программирования они не требуют.

1. Начните с исходной конфигурации приложения Queue Workshop. Поочередно выполняйте операции извлечения и вставки элемента. (Это позволит повторно использовать удаленное ключевое значение без его повторного ввода.) Проследите за тем, как группа из четырех элементов доходит до верха очереди, затем появляется внизу и продолжает двигаться наверх.
2. Используя приложение PriorityQ Workshop, определите позиции маркеров `Front` и `Rear` для заполненной и пустой приоритетной очереди. Почему в приоритетной очереди нельзя использовать циклический перенос, как в обычной?
3. Подумайте над тем, как вы запоминаете события в своей жизни. Бывают ли ситуации, когда воспоминания хранятся по принципу стека? Очереди? Приоритетной очереди?

Программные проекты

В этом разделе читателю предлагаются программные проекты, которые укрепляют понимание материала и демонстрируют практическое применение концепций этой главы.

4.1. Напишите метод класса `Queue` программы queue.java (см. листинг 4.4) для вывода содержимого очереди. Учтите, что задача не сводится к простому выводу содержимого базового массива. Содержимое очереди должно выводиться от первого вставленного элемента до последнего, а пользователь не должен видеть, что последовательность прерывается на границе массива. Будьте внимательны и проследите за тем, чтобы один элемент и содержимое пустой очереди выводились корректно независимо от положения `front` и `rear`.

4.2. Создайте класс `Deque` по описанию деков (двусторонних очередей) в этой главе. Класс должен содержать методы `insertLeft()`, `insertRight()`, `removeLeft()`, `removeRight()`, `isEmpty()` и `isFull()`. Также в нем должна быть реализована поддержка циклического переноса индексов, по аналогии с очередями.

4.3. Напишите реализацию стека на базе класса `Deque` из п. 4.2. Класс стека должен поддерживать те же методы и возможности, что и класс `StackX` в программе `stack.java` (см. листинг 4.1).

4.4. Реализация приоритетной очереди из листинга 4.6 обеспечивает быстрое извлечение высокоприоритетных элементов, но вставка новых элементов выполняется относительно медленно. Напишите программу с видоизмененным классом PriorityQ, быстро выполняющим вставку (за время $O(1)$) с медленным извлечением высокоприоритетного элемента. Включите метод для вывода содержимого приоритетной очереди (см. п. 4.1).

4.5. Очереди часто используются для моделирования потока людей, машин, самолетов, банковских операций и т. д. Напишите программу, моделирующую очередь покупателей в кассы в магазине, на базе класса `Queue` из программы `queue.java` (см. листинг 4.4). Программа должна отображать содержимое сразу нескольких очередей; воспользуйтесь методом `display()` из п. 4.1. Новый покупатель помещается в очередь нажатием клавиши. Вы должны самостоятельно определить, каким образом он будет выбирать очередь. Обслуживание каждого покупателя имеет случайную продолжительность (в зависимости от количества товаров в корзине). Обслуженные покупатели удаляются из очереди. Для простоты течение времени можно моделировать нажатиями клавиш — например, каждое нажатие клавиши соответствует одной минуте. (Конечно, в Java есть и более совершенные средства для работы со временем.)

Глава 5
Связанные списки

В главе 2, «Массивы», было показано, что хранение данных в массивах имеет ряд недостатков. В неупорядоченном массиве поиск выполняется относительно медленно, тогда как в упорядоченном массиве медленно выполняется вставка. Удаление выполняется медленно в обеих разновидностях массивов. Кроме того, размер массива невозможно изменить после его создания.

В этой главе рассматриваются связанные списки — структура данных, которая решает некоторые из этих проблем. Вероятно, связанный список является второй по популярности структурой данных после массива.

Гибкость связанных списков хорошо подходит для многих общих задач хранения данных. Кроме того, связанный список может заменить массив в качестве базы для других структур хранения данных (таких, как стеки и очереди). Более того, связанные списки часто могут использоваться вместо массивов (исключение составляют ситуации с частым произвольным доступом к отдельным элементам по индексу).

Связанные списки не решают всех проблем хранения данных, но они на редкость универсальны, а на концептуальном уровне более просты, чем другие популярные структуры данных (например, деревья). Их достоинства и недостатки будут проанализированы по мере изложения материала.

В этой главе будут рассмотрены простые связанные списки, двусторонние списки, сортированные списки, двусвязные списки, а также списки с итераторами (решение проблемы произвольного доступа к элементам списка). Также мы познакомимся с концепцией абстрактных типов данных (ADT), увидим, как стеки и очереди интерпретируются как типы ADT и как они реализуются на базе связанных списков вместо массивов.

Строение связанного списка

В связанном списке каждый элемент данных встраивается в специальный объект, называемый *элементом списка* (классу, на основе которого создаются такие объекты, часто присваивается имя `Link`). Так как список содержит много однотипных элементов, для них удобно создать отдельный класс, отличный от класса самого связанного списка. Каждый элемент (то есть объект `Link`) содержит ссылку на следующий элемент списка; поле, в котором эта ссылка хранится, обычно называется `next`. Объект списка содержит ссылку на первый элемент `first`. Отношения между объектами в этой архитектуре представлены на рис. 5.1.

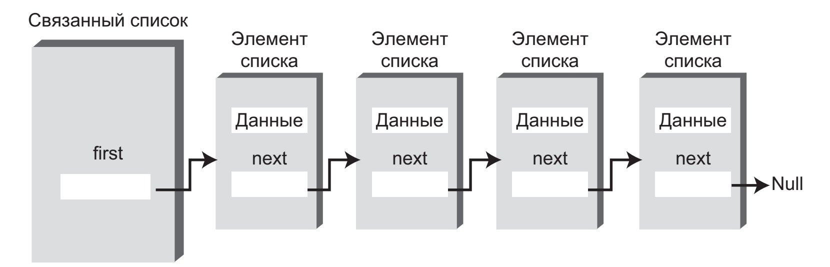


Рис. 5.1. Связанный список

Далее приводится часть определения класса `Link`. В полях класса хранятся данные и ссылка на следующий элемент списка:

```
class Link
{
   public int iData;     // Данные
   public double dData;  // Данные
   public Link next;     // Ссылка на следующий элемент списка
}
```

Подобные определения классов иногда называются *самоотносимыми* (self-referential), потому что объект класса содержит поле (`next` в данном случае) со ссылкой на объект того же типа. В типичном приложении полей данных может быть намного больше. Скажем, запись о работнике фирмы может содержать поля имени, адреса, номера социального страхования, должности, зарплаты, а также другие атрибуты. Вместо отдельных полей часто используется объект класса, содержащего все данные:

```
class Link
   {
   public inventoryItem iI;  // Объект с данными
   public Link next;         // Ссылка на следующий элемент списка
   }
```

Ссылки и базовые типы

Многие читатели смутно представляют себе, как работают ссылки в контексте связанных списков. Давайте разберемся в них поподробнее.

Включение поля типа `Link` в определение класса того же типа на первый взгляд выглядит немного странно. Разве компилятор не запутается? Как он определит, сколько памяти выделить под объект `Link`, если объект содержит сам себя, а компилятор еще не знает, сколько памяти занимает объект `Link`?

Дело в том, что в языке Java объект `Link` не содержит другой объект `Link`. Поле `next` типа `Link` содержит только *ссылку* на другой элемент списка, а не сам объект.

Ссылка представляет собой число, ассоциированное с объектом. Это число соответствует адресу объекта в памяти компьютера, но его конкретное значение вас не интересует; просто считайте, что это «волшебное число» определяет

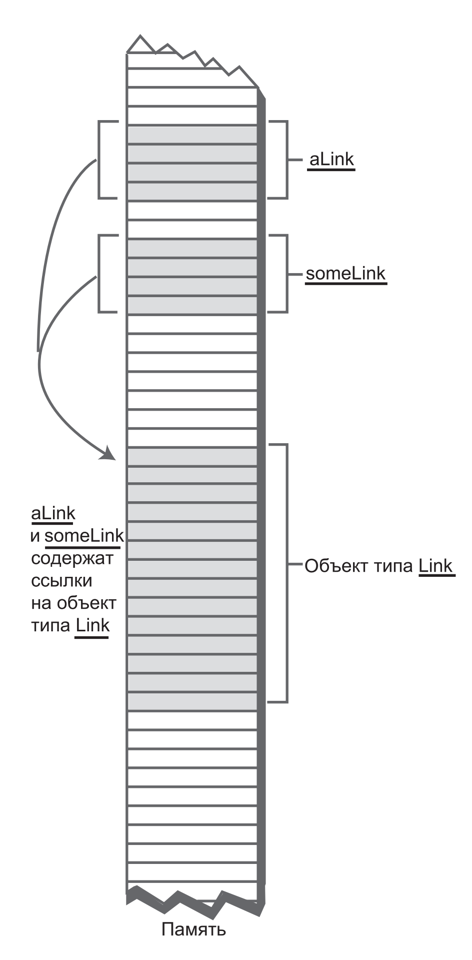


Рис. 5.2. Ссылки и объекты в памяти

местонахождение объекта. Для конкретного компьютера и операционной системы все ссылки, на что бы они ни ссылались, всегда имеют одинаковый размер. Таким образом, компилятор легко вычислит размер поля, а следовательно, сконструирует весь объект `Link`.

Следует учитывать, что в языке Java примитивные типы (`int`, `double` и т. д.) хранятся в памяти не так, как объекты. В полях примитивных типов хранятся не ссылки, а конкретные числовые значения вроде 7 или 3,14159. Определение переменной

```
double salary = 65000.00;
```

резервирует область памяти и размещает в ней число 65000.00. С другой стороны, ссылка на объект вида

```
Link aLink = someLink;
```

помещает в переменную `aLink` ссылку на объект `someLink` типа `Link`. Сам объект `someLink` хранится где-то в другом месте. Команда не перемещает его и даже не создает; объект должен быть создан заранее. Объекты всегда создаются ключевым словом `new`:

```
Link someLink = new Link();
```

Даже в этом случае поле `someLink` содержит не объект, а только ссылку на него. Объект находится где-то в другом месте памяти (рис. 5.2).

Другие языки (такие, как C++) работают с объектами иначе. В C++ поле вида

```
Link next;
```

действительно содержит объект типа `Link`. В C++ невозможно написать определение самоотносимого класса (хотя в класс `Link` можно включить указатель на `Link`; указатель является аналогом ссылки). Программисты C++ должны помнить об этой особенности работы со объектами в Java; на первый взгляд она может показаться неочевидной.

Отношения вместо конкретных позиций

Рассмотрим одно из важнейших отличий между связанными списками и массивами. В массиве каждый элемент всегда занимает конкретную позицию и к нему можно обратиться напрямую по индексу. Все происходит, как при поиске дома на незнакомой улице: нужный дом легко находится по адресу.

В списке конкретный элемент можно найти только одним способом: отследив его по цепочке элементов от начала списка. Поиск больше напоминает отношения между людьми. Допустим, вы спрашиваете Гарри, где находится Боб. Гарри этого не знает, но он думает, что Джейн может вам помочь. Вы идете и спрашиваете Джейн. Она видела, что Боб уходил из офиса с Салли; вы звоните на сотовый телефон Салли. Оказывается, Салли рассталась с Бобом у офиса Питера... В общем, вы поняли. Обратиться к элементу данных напрямую невозможно, для поиска приходится использовать отношения между элементами. Вы начинаете с первого элемента, переходите ко второму, потом к третьему — пока не найдете тот, который вам нужен.

Приложение LinkList Workshop

Приложение LinkList Workshop выполняет три операции со списками. Пользователь может вставить новый элемент, выполнить поиск или удалить элемент данных с заданным ключом. Эти операции не отличаются от операций приложения Array Workshop из главы 2 и хорошо подходят для приложения баз данных общего назначения.

На рис. 5.3 изображено окно приложения LinkList Workshop на момент запуска. В исходном состоянии список состоит из 13 элементов.

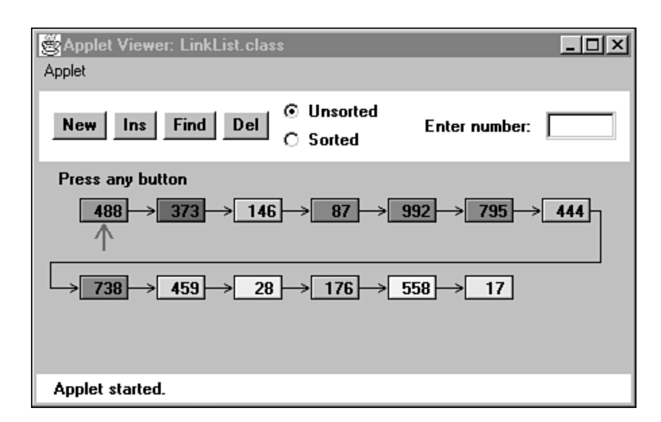


Рис. 5.3. Приложение LinkList Workshop

Вставка

Вам кажется, что 13 — несчастливое число? Вставьте в список новый элемент. При нажатии кнопки `Ins` предлагается ввести значение ключа в диапазоне от 0 до 999. После еще нескольких нажатий в списке создается новый элемент (рис. 5.4).

В этой версии связанного списка новые элементы всегда вставляются в начале списка. Это самое простое решение, хотя, как мы вскоре увидим, новые элементы могут вставляться в любых позициях списка.

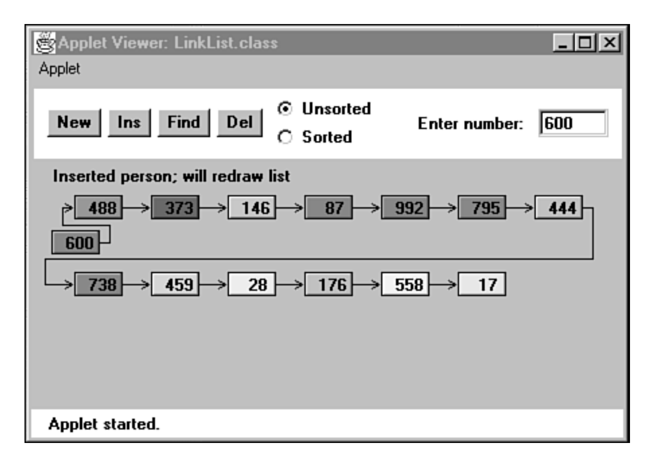


Рис. 5.4. Вставка нового элемента

Последнее нажатие кнопки Ins приводит к перерисовке списка, а только что вставленный элемент появляется в цепочке с другими элементами. При перерисовке программа не выполняет никакие содержательные действия — просто изображение становится более аккуратным.

Поиск

Кнопка Find предназначена для поиска элемента списка с заданным ключом. Введите ключевое значение существующего элемента, желательно откуда-нибудь из середины списка. Вы увидите, как при последующих нажатиях кнопки по списку перемещается красная стрелка, обозначающая текущую позицию поиска. Когда элемент будет найден, появляется подтверждающее сообщение. Если ввести несуществующее ключевое значение, стрелка дойдет до конца списка, после чего приложение сообщит о том, что элемент найти не удалось.

Удаление

Приложение также позволяет удалить элемент с заданным ключом. Введите существующее ключевое значение и несколько раз нажмите Del. Стрелка снова перемещается по списку в поисках нужного элемента. Если поиск будет успешным, приложение просто удаляет этот элемент из цепочки и переводит стрелку от предыдущего элемента прямо к следующему. Так происходит удаление элементов: ссылка в предыдущем элементе заменяется ссылкой на следующий элемент.

При последнем нажатии кнопки изображение перерисовывается, но как и в предыдущем случае, перерисовка всего лишь равномерно распределяет ссылки в окне по эстетическим соображениям; от длины стрелок в работе программы ничего не зависит.

ПРИМЕЧАНИЕ

Приложение LinkList Workshop может создавать как несортированные, так и сортированные списки. По умолчанию создается несортированный список. О том, как использовать приложение для создания сортированного списка, будет рассказано позднее в этой главе.

Простой связанный список

В нашем первом примере `linkList.java` представлен простой связанный список. Эта разновидность списка поддерживает следующие операции:

- Вставка элемента в начале списка.
- Удаление элемента в начале списка.
- Перебор списка для вывода содержимого.

Все операции выполняются относительно просто, поэтому мы начнем с них. (Как вы увидите позднее, эти операции — все, что необходимо для использования связанного списка в качестве основы для реализации стека.)

Но прежде чем разбирать код программы `linkList.java`, мы рассмотрим важнейшие составляющие классов `Link` и `LinkList`.

Класс Link

Вы уже видели часть класса `Link`, предназначенную для хранения данных. Полное определение класса выглядит так:

```
class Link
   {
   public int iData;              // Данные
   public double dData;           // Данные
   public Link next;              // Следующий элемент в списке
// -------------------------------------------------------------
   public Link(int id, double dd) // Конструктор
      {
      iData = id;                 // Инициализация данных
      dData = dd;                 // (‘next’ автоматически
      }                           // присваивается null)
// -------------------------------------------------------------
   public void displayLink()      // Вывод содержимого элемента
      {
      System.out.print("{" + iData + ", " + dData + "} ");
      }
   }  // Конец класса Link
```

Кроме полей данных, класс содержит конструктор и метод `displayLink()` для вывода данных элемента в формате {22, 33.9}. Вероятно, блюстителям чистоты ООП не понравится имя `displayLink()` — они скажут, что метод должен называться просто `display()`. Возможно, короткое имя лучше соответствует духу полиморфизма, но оно усложняет чтение кода. Когда вы встречаете команду вида

```
current.display();
```

вам приходится подолгу вспоминать, что собой представляет current — объект Link, объект LinkList или что-то еще.

Конструктор инициализирует данные. Инициализировать поле next не нужно, потому что в него автоматически записывается null при создании. (Впрочем, для наглядности его можно явно инициализировать null.) Значение null указывает на то, что ссылка не указывает ни на что — до того, как элемент будет связан с другими элементами, дело обстоит именно так.

Поля данных Link (iData и т. д.) объявлены открытыми (public). Если объявить их приватными, нам придется предоставлять открытые методы для работы с ними; это потребует лишнего кода, а листинг станет длиннее и будет хуже читаться. В идеале по соображениям безопасности нам следовало бы ограничить доступ к объектам Link методами класса LinkList. Тем не менее без отношений наследования между этими классами эта задача усложняется. Мы могли бы воспользоваться спецификатором доступа по умолчанию (без указания ключевого слова), чтобы установить для данных пакетный уровень доступа (доступ ограничивается классами того же каталога), но это никак не отразится на работе наших демонстрационных программ, которые в любом случае занимают только один каталог. Спецификатор public по крайней мере ясно показывает, что данные не являются приватными. Вероятно, в более серьезной программе все поля данных класса Link стоило бы объявить с ключевым словом private.

Класс LinkList

Класс LinkList содержит всего один элемент данных: ссылку на первый элемент списка. Ссылка хранится в поле с именем first. Это единственная информация о местонахождении элементов, которая хранится в списке. Остальные элементы отслеживаются по цепочке ссылок next, начиная с элемента first:

```
class LinkList
   {
   private Link first;            // Ссылка на первый элемент списка
// -------------------------------------------------------------
   public void LinkList()         // Конструктор
      {
      first = null;               // Список пока не содержит элементов
      }
// -------------------------------------------------------------
   public boolean isEmpty()       // true, если список пуст
      {
      return (first==null);
      }
// -------------------------------------------------------------
   //   ...   Другие методы
}
```

Конструктор LinkList инициализирует first значением null. На самом деле явная инициализация не обязательна, потому что, как упоминалось ранее, это делается автоматически при создании объекта. Тем не менее явная инициализация в конструкторе четко показывает исходное состояние first.

Когда `first` содержит `null`, список не содержит ни одного элемента. Если бы в списке был хоть один элемент, в `first` хранилась бы ссылка на него. Метод `isEmpty()` использует этот факт для проверки отсутствия элементов в списке.

Метод insertFirst()

Метод `insertFirst()` класса `LinkList` вставляет новый элемент в начало списка. В этой позиции вставка выполняется проще всего, потому что `first` уже указывает на первый элемент. Чтобы вставить в список новый элемент, достаточно присвоить полю `next` созданного объекта ссылку на предыдущий первый элемент, а затем изменить поле `first` так, чтобы оно указывало на только что вставленный элемент. Ситуация показана на рис. 5.5.

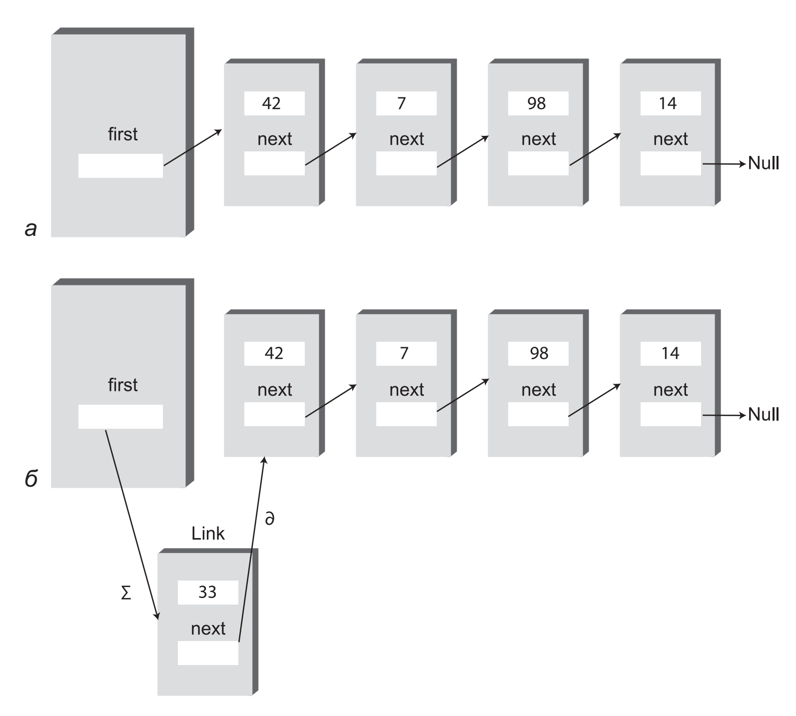


Рис. 5.5. Вставка нового элемента: *а* — до вставки; *б* — после вставки

Выполнение метода `insertFirst()` начинается с создания нового элемента на основании данных, переданных в аргументах. Затем ссылки изменяются так, как говорилось ранее:

```
// Вставка элемента в начало списка
public void insertFirst(int id, double dd)
   {                            // Создание нового элемента
   Link newLink = new Link(id, dd);
   newLink.next = first;        // newLink --> старое значение first
   first = newLink;             // first --> newLink
   }
```

Стрелки --> в комментариях к двум последним командам означают, что элемент (или поле first) связывается со следующим элементом в цепочке. (В двусвязных списках также устанавливаются обратные связи между элементами, обозначаемые стрелками <--.) Сравните эти две команды с рис. 5.5. Убедитесь в том, что вы понимаете, как изменяются ссылки. Подобные манипуляции со ссылками лежат в основе всех алгоритмов связанных списков.

Метод deleteFirst()

Метод deleteFirst() является противоположностью insertFirst(). Он отсоединяет первый элемент, для чего в поле first заносится ссылка на второй элемент (который находится по значению поля next в первом элементе):

```
public Link deleteFirst()      // Удаление первого элемента
   {                           // (предполагается, что список не пуст)
   Link temp = first;          // Сохранение ссылки
   first = first.next;         // Удаление: first-->ссылка на второй элемент
   return temp;                // Метод возвращает ссылку
   }                           // на удаленный элемент
```

Вторая команда — все, что необходимо для удаления первого элемента из списка. Наш метод также возвращает ссылку на удаленный элемент для удобства пользователя связанного списка, поэтому он сохраняет ссылку во временной переменной temp перед удалением и возвращает значение temp. На рис. 5.6 показано, как изменяется значение first при удалении объекта.

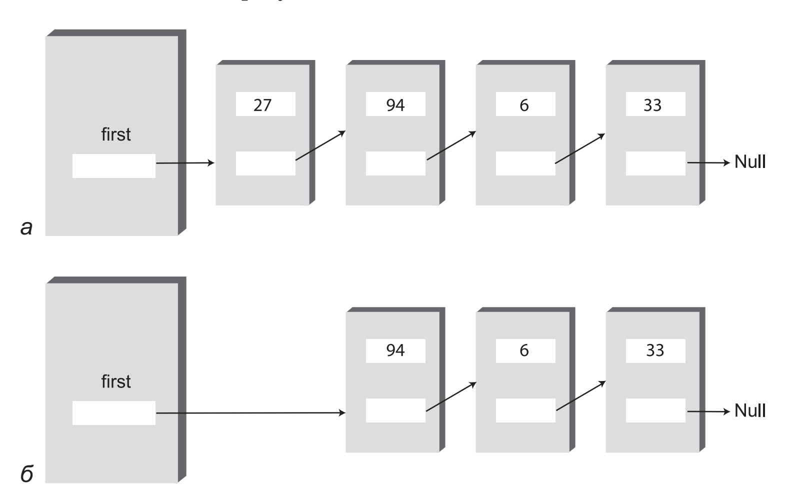


Рис. 5.6. Удаление элемента: *а* — до удаления; *б* — после удаления

В C++ и других аналогичных языках программисту придется также побеспокоиться об удалении самого элемента, отсоединенного от списка. Элемент находится где-то в памяти, но ссылки на него отсутствуют. Что с ним станет? В Java он будет уничтожен когда-нибудь в будущем уборщиком мусора; вас это уже не касается.

Обратите внимание: метод deleteFirst() предполагает, что список не пуст. Прежде чем вызывать его, программа должна проверить состояние списка методом isEmpty().

Метод displayList()

Чтобы вывести содержимое списка, мы начинаем со ссылки first и перемещаемся от элемента к элементу. Переменная current последовательно указывает (а точнее, ссылается) на каждый элемент списка. Сначала она содержит first, то есть ссылку на первый элемент списка. Команда

```
current = current.next;
```

присваивает current ссылку на следующий элемент списка, так как именно эта ссылка хранится в поле next каждого элемента. Полный код метода displayList():

```
public void displayList()
   {
   System.out.print("List (first-->last): ");
   Link current = first;       // От начала списка
   while(current != null)      // Перемещение до конца списка
      {
      current.displayLink();   // Вывод данных
      current = current.next;  // Переход к следующему элементу
      }
   System.out.println("");
   }
```

У последнего элемента списка поле next содержит null вместо указателя на другой элемент. Как в это поле попало значение null? Поле было инициализировано им при создании объекта, а другие значения в него не записывались, потому что этот элемент всегда находился в конце списка. Это условие прерывает выполнение цикла while при достижении конца списка. На рис. 5.7 показано, как переменная current перемещается по списку.

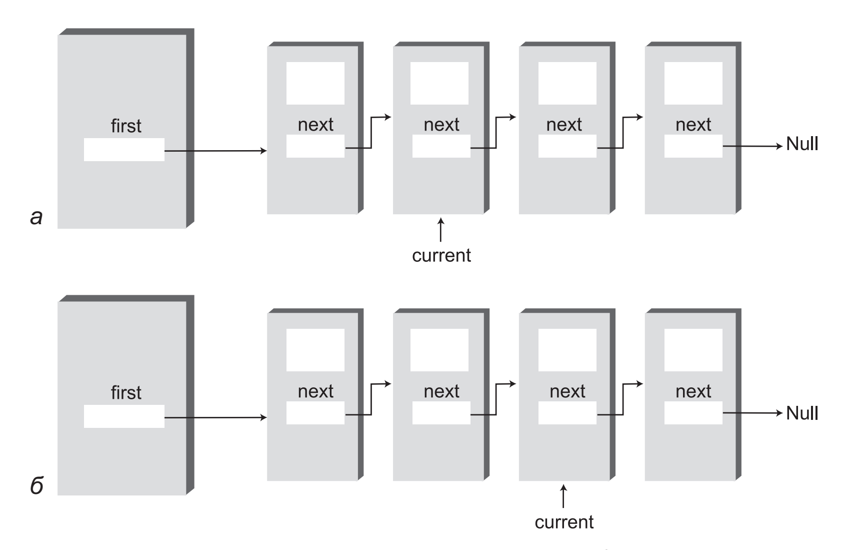


Рис. 5.7. Перемещение по списку: *а* — до current = current.next; *б* — после current = current.next

На каждой итерации метод displayList() вызывает метод displayLink() для вывода данных текущего элемента.

Программа linkList.java

В листинге 5.1 приведен полный код программы `linkList.java`. Все ее компоненты, кроме метода `main()`, уже были описаны ранее.

Листинг 5.1. Программа linkList.java

```
// linkList.java
// Работа со связанным списком
// Запуск программы: C>java LinkListApp
////////////////////////////////////////////////////////////////
class Link
   {
   public int iData;              // Данные (ключ)
   public double dData;           // Данные
   public Link next;              // Следующий элемент в списке
// -------------------------------------------------------------
   public Link(int id, double dd) // Конструктор
      {
      iData = id;                 // Инициализация данных
      dData = dd;                 // ('next' автоматически
      }                           //  присваивается null)
// -------------------------------------------------------------
   public void displayLink()      // Вывод содержимого элемента
      {
      System.out.print("{" + iData + ", " + dData + "} ");
      }
   }  // Конец класса Link
////////////////////////////////////////////////////////////////
class LinkList
   {
   private Link first;            // Ссылка на первый элемент списка
// -------------------------------------------------------------
   public LinkList()              // Конструктор
      {
      first = null;               // Список пока не содержит элементов
      }
// -------------------------------------------------------------
   public boolean isEmpty()       // true, если список пуст
      {
      return (first==null);
      }
// -------------------------------------------------------------
// Вставка элемента в начало списка
   public void insertFirst(int id, double dd)
      {                           // Создание нового элемента
      Link newLink = new Link(id, dd);
      newLink.next = first;       // newLink --> старое значение first
      first = newLink;            // first --> newLink
      }
```

```
// -------------------------------------------------------------
   public Link deleteFirst()   // Удаление первого элемента
      {                        // (предполагается, что список не пуст)
      Link temp = first;       // Сохранение ссылки
      first = first.next;      // Удаление: first-->ссылка на второй элемент
      return temp;             // Метод возвращает ссылку
      }                        // на удаленный элемент
// -------------------------------------------------------------
   public void displayList()
      {
      System.out.print("List (first-->last): ");
      Link current = first;       // От начала списка
      while(current != null)      // Перемещение до конца списка
         {
         current.displayLink();   // Вывод данных
         current = current.next;  // Переход к следующему элементу
         }
      System.out.println("");
      }
// -------------------------------------------------------------
   }  // Конец класса LinkList
////////////////////////////////////////////////////////////////
class LinkListApp
   {
   public static void main(String[] args)
      {
      LinkList theList = new LinkList();  // Создание нового списка
      theList.insertFirst(22, 2.99);      // Вставка четырех элементов
      theList.insertFirst(44, 4.99);
      theList.insertFirst(66, 6.99);
      theList.insertFirst(88, 8.99);

      theList.displayList();              // Вывод содержимого списка

      while( !theList.isEmpty() )         // Пока остаются элементы,
         {
         Link aLink = theList.deleteFirst();   // Удаление элемента
         System.out.print("Deleted ");         // Вывод удаленного элемента
         aLink.displayLink();
         System.out.println("");
         }
      theList.displayList();              // Вывод содержимого списка
      }
   }  // Конец класса LinkListApp
////////////////////////////////////////////////////////////////
```

Метод main() создает новый список, вставляет в него четыре элемента при помощи метода insertFirst() и выводит заполненный список. Затем в цикле while элементы последовательно удаляются из списка методом deleteFirst(), пока список

не опустеет, после чего программа выводит пустой список. Результат выполнения `linkList.java`:

```
List (first-->last): {88, 8.99} {66, 6.99} {44, 4.99} {22, 2.99}
Deleted {88, 8.99}
Deleted {66, 6.99}
Deleted {44, 4.99}
Deleted {22, 2.99}
List (first-->last):
```

Поиск и удаление заданных элементов

Следующая программа дополняется новыми методами для поиска и удаления из списка элемента с заданным значением ключа. Если добавить к ним вставку в начале списка, мы получаем набор операций из приложения LinkList Workshop. Полный код программы `linkList2.java` приведен в листинге 5.2.

Листинг 5.2. Программа linkList2.java

```
// linkList2.java
// Работа со связанным списком
// Запуск программы: C>java LinkList2App
////////////////////////////////////////////////////////////////
class Link
   {
   public int iData;              // Данные (ключ)
   public double dData;           // Данные
   public Link next;              // Следующий элемент в списке
// -------------------------------------------------------------
   public Link(int id, double dd) // Конструктор
      {
      iData = id;
      dData = dd;
      }
// -------------------------------------------------------------
   public void displayLink()      // Вывод содержимого элемента
      {
      System.out.print("{" + iData + ", " + dData + "} ");
      }
   }  // Конец класса Link
////////////////////////////////////////////////////////////////
class LinkList
   {
   private Link first;            // Ссылка на первый элемент списка
// -------------------------------------------------------------
   public LinkList()              // Конструктор
      {
      first = null;               // Список пока не содержит элементов
      }
```

```
// -------------------------------------------------------------
   public void insertFirst(int id, double dd)
      {                           // Создание нового элемента
      Link newLink = new Link(id, dd);
      newLink.next = first;       // newLink --> старое значение first
      first = newLink;            // first --> newLink
   }
// -------------------------------------------------------------
   public Link find(int key)      // Поиск элемента с заданным ключом
      {                           // (предполагается, что список не пуст)
      Link current = first;              // Начиная с 'first'
      while(current.iData != key)        // Пока совпадение не найдено
         {
         if(current.next == null)        // Если достигнут конец списка
            return null;                 // и совпадение не найдено
         else                            // Если еще остались элементы
            current = current.next;      // Перейти к следующему элементу
         }
      return current;                    // Совпадение обнаружено
      }
// -------------------------------------------------------------
   public Link delete(int key)    // Удаление элемента с заданным ключом
      {                           // (предполагается, что список не пуст)
      Link current = first;              // Поиск элемента
      Link previous = first;
      while(current.iData != key)
         {
         if(current.next == null)
            return null;                 // Элемент не найден
         else
            {
            previous = current;          // Перейти к следующему элементу
            current = current.next;
            }
         }                               // Совпадение найдено
      if(current == first)               // Если первый элемент,
         first = first.next;             // изменить first
      else                               // В противном случае
         previous.next = current.next;   // обойти его в списке
      return current;
      }
// -------------------------------------------------------------
   public void displayList()      // Вывод содержимого списка
      {
      System.out.print("List (first-->last): ");
      Link current = first;       // От начала списка
      while(current != null)      // Перемещение до конца списка
         {
         current.displayLink();   // Вывод данных
```

продолжение ⇨

Листинг 5.2 (*продолжение*)

```
        current = current.next;  // Переход к следующему элементу
        }
      System.out.println("");
      }
// -------------------------------------------------------------
   }  // Конец класса LinkList
////////////////////////////////////////////////////////////////
class LinkList2App
   {
   public static void main(String[] args)
      {
      LinkList theList = new LinkList();  // Создание нового списка

      theList.insertFirst(22, 2.99);      // Вставка 4 элементов
      theList.insertFirst(44, 4.99);
      theList.insertFirst(66, 6.99);
      theList.insertFirst(88, 8.99);

      theList.displayList();              // Вывод содержимого списка

      Link f = theList.find(44);          // Поиск элемента
      if( f != null)
         System.out.println("Found link with key " + f.iData);
      else
         System.out.println("Can't find link");

      Link d = theList.delete(66);        // Удаление элемента
      if( d != null )
         System.out.println("Deleted link with key " + d.iData);
      else
         System.out.println("Can't delete link");

      theList.displayList();              // Вывод содержимого списка
      }
   }  // Конец класса LinkList2App
////////////////////////////////////////////////////////////////
```

Метод main() создает список, вставляет четыре элемента и выводит содержимое полученного списка. Затем он ищет элемент с ключом 44, удаляет элемент с ключом 66 и снова выводит измененный список. Результат работы программы:

```
List (first-->last): {88, 8.99} {66, 6.99} {44, 4.99} {22, 2.99}
Found link with key 44
Deleted link with key 66
List (first-->last): {88, 8.99} {44, 4.99} {22, 2.99}
```

Метод find()

Принцип работы метода find() напоминает метод displayList() из программы linkList.java. Ссылка current, изначально содержащая указатель first, перебирает ссылки по цепочке, для чего ей многократно присваивается current.next. Для каждого элемента списка find() проверяет, совпадает ли ключ данного элемента с искомым. Если ключи совпадают, метод возвращает ссылку на текущий элемент. Если find() доходит до конца списка, так и не обнаружив нужной ссылки, возвращается null.

Метод delete()

Для поиска удаляемого элемента метод delete() использует тот же способ, что и метод find(). Однако этот метод должен хранить ссылку не только на текущий элемент списка (current), но и на предыдущий элемент (previous). Это необходимо из-за того, что при удалении текущего элемента метод должен связать предыдущий элемент со следующим (рис. 5.8). Чтобы обратиться к предыдущему элементу, необходимо сохранить ссылку на него.

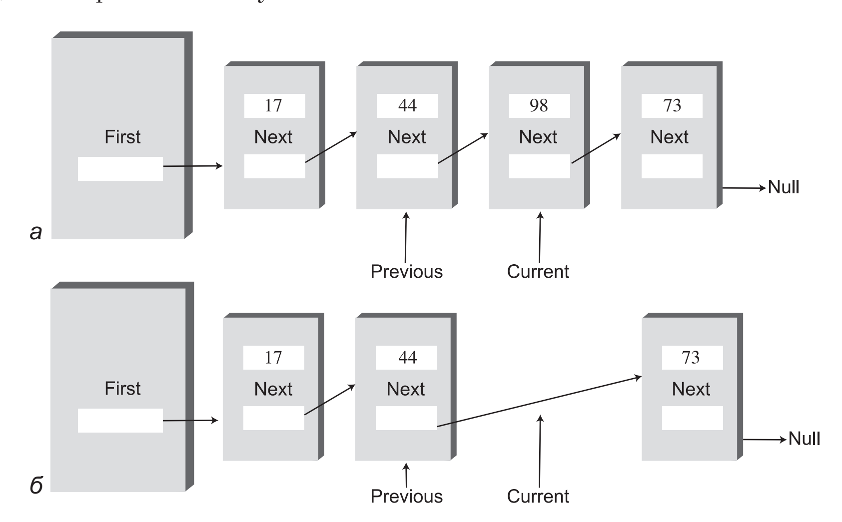


Рис. 5.8. Удаление заданного элемента: *а* — до удаления; *б* — после удаления

На каждой итерации цикла while, непосредственно перед занесением в current значения current.next, в поле previous заносится текущее значение current. Таким образом, previous указывает на элемент списка, предшествующий current.

Чтобы удалить текущий элемент после обнаружения совпадения, полю next предыдущего элемента присваивается значение next текущего элемента. Необходимо учесть особый случай, когда текущий элемент является первым в списке, потому что ссылка на него хранится в поле first объекта LinkList, а не в другом элементе списка. В этом случае удаление элемента осуществляется занесением

в first ссылки из first.next, как в методе deleteFirst() программы linkList.java. Фрагмент, в котором проверяются эти две возможности, выглядит так:

```
                                    // Совпадение найдено
if(current == first)                // Если первый элемент,
   first = first.next;              // изменить first
else                                // В противном случае
   previous.next = current.next;    // обойти его в списке
```

Другие методы

Мы рассмотрели методы для вставки и удаления элементов в начале списка, а также поиска и удаления элемента с заданным ключом. Нетрудно представить себе и другие полезные методы списков. Например, метод insertAfter() может искать элемент с заданным ключом и вставлять новый элемент после него. Пример такого метода будет приведен при обсуждении списковых итераторов в конце этой главы.

Двусторонние списки

Двусторонний список похож на обычный связанный список с одной дополнительной возможностью: в нем хранится ссылка не только на первый, но и на последний элемент. Пример показан на рис. 5.9.

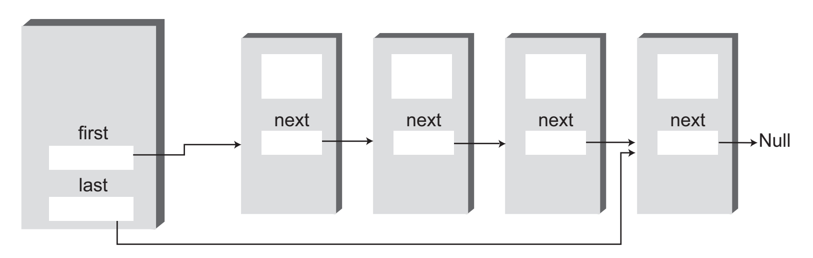


Рис. 5.9. Двусторонний список

Наличие ссылки на последний элемент позволяет вставлять новые элементы не только в начале, но и в конце списка. Конечно, новый элемент можно вставить в конце и обычного односвязного списка, перебирая все его содержимое до последнего элемента, но такое решение неэффективно.

Благодаря возможности быстрого обращения к последнему элементу (вместе с первым) двусторонний список хорошо подходит для некоторых ситуаций, в которых односторонний список недостаточно эффективен. Например, двусторонний список может использоваться для реализации очереди; пример будет рассмотрен в следующем разделе.

В листинге 5.3 приведена программа firstLastList.java, демонстрирующая работу с двусторонним списком. (Кстати говоря, не путайте двусторонний список с двусвязным списком, который также будет рассмотрен позднее в этой главе.)

Листинг 5.3. Программа firstLastList.java

```
// firstLastList.java
// Работа с двусвязным списком
// Запуск программы: C>java FirstLastApp
////////////////////////////////////////////////////////////////
class Link
   {
   public long dData;                 // Данные
   public Link next;                  // Следующий элемент в списке
// -------------------------------------------------------------
   public Link(long d)                // Конструктор
      { dData = d; }
// -------------------------------------------------------------
   public void displayLink()          // Вывод содержимого элемента
      { System.out.print(dData + " "); }
// -------------------------------------------------------------
   }  // Конец класса Link
////////////////////////////////////////////////////////////////
class FirstLastList
   {
   private Link first;               // Ссылка на первый элемент
   private Link last;                // Ссылка на последний элемент
// -------------------------------------------------------------
   public FirstLastList()            // Конструктор
      {
      first = null;                  // Список пока не содержит элементов
      last = null;
      }
// -------------------------------------------------------------
   public boolean isEmpty()          // true, если список пуст
      { return first==null; }
// -------------------------------------------------------------
   public void insertFirst(long dd)  // Вставка элемента в начало списка
      {
      Link newLink = new Link(dd);   // Создание нового элемента
      if( isEmpty() )                // Если список пуст,
         last = newLink;             // newLink <-- last
      newLink.next = first;          // newLink --> старое значение first
      first = newLink;               // first --> newLink
   }
// -------------------------------------------------------------
   public void insertLast(long dd)   // Вставка элемента в конец списка
      {
      Link newLink = new Link(dd);   // Создание нового элемента
      if( isEmpty() )                // Если список пуст,
         first = newLink;            // first --> newLink
      else
         last.next = newLink;        // Старое значение last --> newLink
```

продолжение ⇨

Листинг 5.3 (*продолжение*)

```
      last = newLink;               // newLink <-- last
      }
// -------------------------------------------------------------
   public long deleteFirst()        // Удаление первого элемента списка
      {                             // (предполагается, что список не пуст)
      long temp = first.dData;
      if(first.next == null)        // Если только один элемент
         last = null;               // null <-- last
      first = first.next;           // first --> старое значение next
      return temp;
      }
// -------------------------------------------------------------
   public void displayList()
      {
      System.out.print("List (first-->last): ");
      Link current = first;         // От начала списка
      while(current != null)        // Перемещение до конца списка
         {
         current.displayLink();     // Вывод данных
         current = current.next;    // Переход к следующему элементу
         }
      System.out.println("");
      }
// -------------------------------------------------------------
   }  // Конец класса FirstLastList
////////////////////////////////////////////////////////////////
class FirstLastApp
   {
   public static void main(String[] args)
      {                             // Создание нового списка
      FirstLastList theList = new FirstLastList();

      theList.insertFirst(22);      // Вставка в начало списка
      theList.insertFirst(44);
      theList.insertFirst(66);

      theList.insertLast(11);       // Вставка в конец списка
      theList.insertLast(33);
      theList.insertLast(55);

      theList.displayList();        // Вывод содержимого списка

      theList.deleteFirst();        // Удаление первых двух элементов
      theList.deleteFirst();

      theList.displayList();        // Повторный вывод
      }
   }  // Конец класса FirstLastApp
////////////////////////////////////////////////////////////////
```

Для простоты в этой программе количество полей данных в каждом элементе сокращено до одного. Это упрощает вывод содержимого элемента. (Не забудьте, что в серьезной программе количество полей данных обычно намного больше — или элемент содержит ссылку на другой объект с множеством полей данных.)

Программа вставляет три элемента в начало списка, потом еще три элемента в конец списка и выводит полученный список. Затем первые два элемента удаляются, а список выводится заново. Результат выглядит так:

```
List (first-->last): 66 44 22 11 33 55
List (first-->last): 22 11 33 55
```

Обратите внимание: повторная вставка в начало списка изменяет порядок элементов, тогда как при повторной вставке в конец списка этот порядок сохраняется.

Класс двустороннего списка называется `FirstLastList`. Как упоминалось ранее, два поля данных, `first` и `last`, содержат ссылки соответственно на первый и последний элементы списка. Если список содержит всего один элемент, то на него ссылаются оба поля, а при отсутствии элементов оба поля содержат `null`.

Класс содержит новый метод `insertLast()` для вставки нового элемента в конец списка. В процессе вставки метод сохраняет ссылку на новый элемент в `last.next` и в `last` (рис. 5.10).

Методы вставки и удаления аналогичны методами одностороннего списка. Впрочем, обоим методам вставки приходится проверять особый случай, когда список пуст до выполнения вставки. Иначе говоря, если вызов `isEmpty()` возвращает `true`, то метод `insertFirst()` должен присвоить `last` ссылку на новый элемент, а метод `insertLast()` должен присвоить эту ссылку `first`.

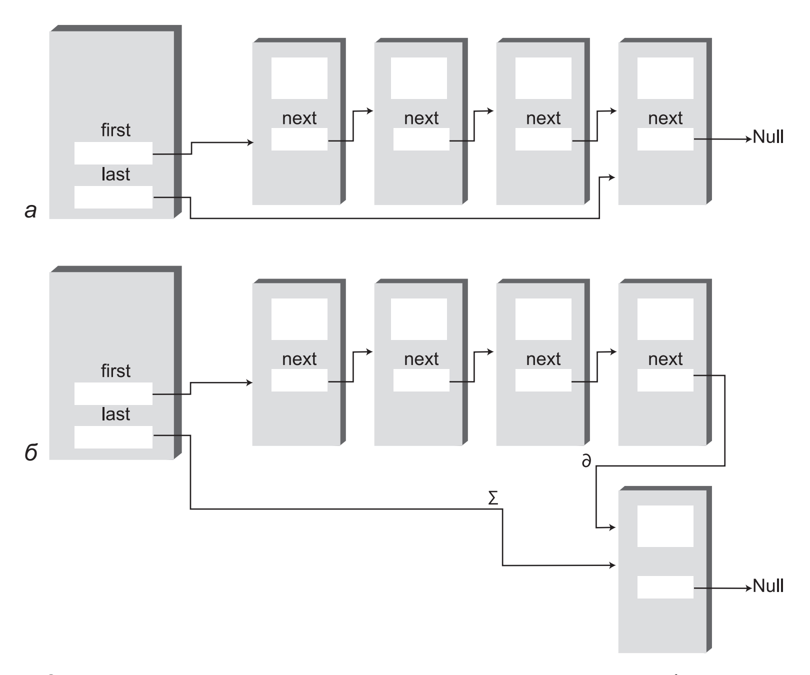


Рис. 5.10. Вставка нового элемента в конец списка: *а* — до вставки; *б* — после вставки

При вставке в начало списка методом insertFirst() в поле first заносится ссылка на новый элемент; при вставке в конце списка методом insertLast() ссылка на новый элемент заносится в поле last. При удалении в начале списка также существует особый случай с удалением последнего элемента в списке (в поле last в этом случае необходимо записать null).

К сожалению, переход от обычного списка к двустороннему не упрощает удаление последнего элемента — ведь в вашем распоряжении нет ссылки на предпоследний элемент, в поле next которого необходимо записать null. Для простого удаления последнего элемента необходимы двусвязные списки, которые мы вскоре рассмотрим. (Конечно, для получения предпоследнего элемента также можно перебрать весь список, но это неэффективно.)

Эффективность связанных списков

Вставка и удаление в начале связанного списка выполняются очень быстро. Операция сводится к изменению одной или двух ссылок, выполняемому за время $O(1)$.

Поиск, удаление и вставка рядом с конкретным элементом требует перебора в среднем половины элементов списка, для которого необходимо $O(N)$ сравнений. У массива эти операции тоже выполняются за время $O(N)$, но связанный список все равно работает быстрее, потому что он не требует перемещения элементов при вставке или удалении. Повышение эффективности может быть весьма значительным, особенно если копирование занимает существенно больше времени, чем сравнение.

Другое важное преимущество связанных списков перед массивами заключается в том, что связанный список всегда использует ровно столько памяти, сколько необходимо, и может расширяться вплоть до всей доступной памяти. Фиксация размера массива при создании обычно приводит либо к неэффективному использованию памяти (если массив слишком велик) или исчерпанию всей доступной памяти (если массив слишком мал). Векторы (динамически расширяемые массивы) до определенной степени решают эту проблему, но они обычно расширяются с фиксированным приращением, например размер массива удваивается перед переполнением. Впрочем, даже это решение не так эффективно использует память, как связанный список.

Абстрактные типы данных

В этом разделе рассматривается тема более общая, чем связанные списки: абстрактные типы данных (ADT, Abstract Data Types). Что такое ADT? Упрощенно говоря, это подход к описанию структуры данных: внимание сосредоточено на том, *что* делает данная структура, а не на том, *как* она это делает.

Стеки и очереди являются примерами абстрактных типов данных. Мы уже видели, что и стеки, и очереди могут быть реализованы на базе массивов. Прежде чем возвращаться к обсуждению абстрактных типов данных, давайте посмотрим,

как стеки и очереди реализуются на базе связанных списков. Это обсуждение продемонстрирует «абстрактную» природу стеков и очередей, то есть возможность анализа их поведения независимо от реализации.

Реализация стека на базе связанного списка

В реализации из главы 4, «Стеки и очереди», данные стека хранились в обычном массиве Java. Операции стека push() и pop() фактически выполнялись операциями массива, такими как

```
arr[++top] = data;
```

и

```
data = arr[top--];
```

(соответственно вставка и извлечение данных из массива).

Данные стека также можно хранить и в связанном списке. В этом случае push() и pop() реализуются такими операциями, как

```
theList.insertFirst(data)
```

и

```
data = theList.deleteFirst()
```

Пользователь класса вызывает push() и pop() для вставки и удаления элементов. При этом он не знает (да ему и не нужно знать), как реализован стек — на базе массива или на базе связанного списка. В листинге 5.4 представлена возможная реализация класса стека LinkStack, в которой вместо массива используется класс LinkList.

Листинг 5.4. Программа linkStack.java

```
// linkStack.java
// Реализация стека на базе связанного списка
// Запуск программы: C>java LinkStackApp
////////////////////////////////////////////////////////////////
class Link
   {
   public long dData;             // Данные
   public Link next;              // Следующий элемент в списке
// -------------------------------------------------------------
   public Link(long dd)           // Конструктор
      { dData = dd; }
// -------------------------------------------------------------
   public void displayLink()      // Вывод содержимого элемента
      { System.out.print(dData + " "); }
      }  // Конце класса Link
////////////////////////////////////////////////////////////////
class LinkList
   {
   private Link first;            // Ссылка на первый элемент в списке
```

продолжение ⇨

Листинг 5.4 (*продолжение*)

```
// -------------------------------------------------------------
   public LinkList()              // Конструктор
      { first = null; }           // Список пока не содержит элементов
// -------------------------------------------------------------
   public boolean isEmpty()       // true, если список пуст
      { return (first==null); }
// -------------------------------------------------------------
   public void insertFirst(long dd) // Вставка элемента в начало списка
      {                           // Создание нового элемента
      Link newLink = new Link(dd);
      newLink.next = first;       // newLink --> старое значение first
      first = newLink;            // first --> newLink
      }
// -------------------------------------------------------------
   public long deleteFirst()  // Удаление первого элемента
      {                       // (предполагается, что список не пуст)
      Link temp = first;      // Сохранение ссылки
      first = first.next;     // Удаление: first-->ссылка на второй элемент
      return temp.dData;      // Метод возвращает данные
      }                       // удаленного элемента
// -------------------------------------------------------------
   public void displayList()
      {
      Link current = first;       // От начала списка
      while(current != null)      // Перемещение до конца списка
         {
         current.displayLink();   // Вывод данных
         current = current.next;  // Переход к следующему элементу
         }
      System.out.println("");
      }
// -------------------------------------------------------------
   }  // Конец класса LinkList
////////////////////////////////////////////////////////////////
class LinkStack
   {
   private LinkList theList;
//--------------------------------------------------------------
   public LinkStack()             // Конструктор
      {
      theList = new LinkList();
      }
//--------------------------------------------------------------
   public void push(long j)     // Размещение элемента на вершине стека
      {
      theList.insertFirst(j);
      }
//--------------------------------------------------------------
```

```
   public long pop()            // Извлечение элемента с вершины стека
      {
      return theList.deleteFirst();
      }
//--------------------------------------------------------------
   public boolean isEmpty()       // true, если стек пуст
      {
      return ( theList.isEmpty() );
      }
//--------------------------------------------------------------
   public void displayStack()
      {
      System.out.print("Stack (top-->bottom): ");
      theList.displayList();
      }
//--------------------------------------------------------------
   }  // Конец класса LinkStack
////////////////////////////////////////////////////////////////
class LinkStackApp
   {
   public static void main(String[] args)
      {
      LinkStack theStack = new LinkStack(); // Создание стека

      theStack.push(20);                    // Вставка элементов
      theStack.push(40);

      theStack.displayStack();              // Вывод содержимого стека

      theStack.push(60);                    // Вставка элементов
      theStack.push(80);

      theStack.displayStack();              // Вывод содержимого стека

      theStack.pop();                       // Извлечение элементов
      theStack.pop();

      theStack.displayStack();              // Вывод содержимого стека
      }
   }  // Конец класса LinkStackApp
////////////////////////////////////////////////////////////////
```

Метод main() создает объект стека, заносит в него два элемента, выводит содержимое стека, заносит еще два элемента, после чего снова выводит содержимое стека. Напоследок он извлекает два элемента и выводит содержимое стека в третий раз. Результат выполнения программы:

```
Stack (top-->bottom): 40 20
Stack (top-->bottom): 80 60 40 20
Stack (top-->bottom): 40 20
```

Обратите внимание на общую структуру программы. Метод main() в классе LinkStackApp работает только с классом LinkStack, а класс LinkStack работает только с классом LinkList. Какие-либо непосредственные взаимодействия между main() и классом LinkList отсутствуют.

А именно, когда метод main() вызывает операцию push() класса LinkStack, этот метод в свою очередь вызывает метод insertFirst() класса LinkList для выполнения фактической вставки данных. Аналогичным образом pop() вызывает deleteFirst() для удаления элемента, а displayStack() вызывает displayList() для вывода содержимого стека. С точки зрения пользователя класса, программирующего main(), использование списковой реализации LinkStack ничем не отличается от использования реализации на базе массива из программы stack.java (см. листинг 4.1) в главе 4.

Реализация очереди на базе связанного списка

Рассмотрим другой пример реализации ADT. В листинге 5.5 представлена очередь, реализованная на базе двустороннего связанного списка.

Листинг 5.5. Программа linkQueue.java

```
// linkQueue.java
// Реализация очереди на базе двустороннего списка
// Запуск программы: C>java LinkQueueApp
////////////////////////////////////////////////////////////////
class Link
   {
   public long dData;                // Данные
   public Link next;                 // Следующий элемент в списке
// -------------------------------------------------------------
   public Link(long d)               // Конструктор
      { dData = d; }
// -------------------------------------------------------------
   public void displayLink()         // Вывод содержимого элемента
      { System.out.print(dData + " "); }
// -------------------------------------------------------------
   }  // Конец класса Link
////////////////////////////////////////////////////////////////
class FirstLastList
   {
   private Link first;               // Ссылка на первый элемент
   private Link last;                // Ссылка на последний элемент
// -------------------------------------------------------------
   public FirstLastList()            // Конструктор
      {
      first = null;                  // Список пока не содержит элементов
      last = null;
      }
// -------------------------------------------------------------
```

```
   public boolean isEmpty()          // true, если список пуст
      { return first==null; }
// -------------------------------------------------------------
   public void insertLast(long dd)   // Вставка элемента в конец списка
      {
      Link newLink = new Link(dd);   // Создание нового элемента
      if( isEmpty() )                // Если список пуст,
         first = newLink;            // first --> newLink
      else
         last.next = newLink;        // Старое значение last --> newLink
      last = newLink;                // newLink <-- last
      }
// -------------------------------------------------------------
   public long deleteFirst()         // Удаление первого элемента
      {                              // (предполагается, что список не пуст)
      long temp = first.dData;
      if(first.next == null)         // Сохранение ссылки
         last = null;                // null <-- last
      first = first.next;            // first --> старое значение next
      return temp;
      }
// -------------------------------------------------------------
   public void displayList()
      {
      Link current = first;          // От начала списка
      while(current != null)         // Перемещение до конца списка
         {
         current.displayLink();      // Вывод данных
         current = current.next;     // Переход к следующему элементу
         }
      System.out.println("");
   }
// -------------------------------------------------------------
   }  // Конец класса FirstLastList
////////////////////////////////////////////////////////////////
class LinkQueue
   {
   private FirstLastList theList;
//--------------------------------------------------------------
   public LinkQueue()                // Конструктор
      { theList = new FirstLastList(); }  // Создание 2-стороннего списка
//--------------------------------------------------------------
   public boolean isEmpty()          // true, если очередь пуста
      { return theList.isEmpty(); }
//--------------------------------------------------------------
   public void insert(long j)        // Вставка элемента в конец очереди
      { theList.insertLast(j); }
//--------------------------------------------------------------
   public long remove()              // Удаление элемента в начале очереди
      {  return theList.deleteFirst();  }
```

продолжение ⇗

Листинг 5.5 (*продолжение*)

```
//--------------------------------------------------------------
   public void displayQueue()
      {
      System.out.print("Queue (front-->rear): ");
      theList.displayList();
      }
//--------------------------------------------------------------
   }  // Конец класса LinkQueue
////////////////////////////////////////////////////////////////
class LinkQueueApp
   {
   public static void main(String[] args)
      {
      LinkQueue theQueue = new LinkQueue();
      theQueue.insert(20);                 // Вставка элементов
      theQueue.insert(40);

      theQueue.displayQueue();             // Вывод содержимого очереди

      theQueue.insert(60);                 // Вставка элементов
      theQueue.insert(80);

      theQueue.displayQueue();             // Вывод содержимого очереди

      theQueue.remove();                   // Удаление элементов
      theQueue.remove();

      theQueue.displayQueue();             // Вывод содержимого очереди
      }
////////////////////////////////////////////////////////////////
```

Программа создает очередь, вставляет в нее два элемента, потом еще два элемента и удаляет два элемента. После выполнения каждой группы операций выводится текущее содержимое очереди. Результат выполнения программы:

```
Queue (front-->rear): 20 40
Queue (front-->rear): 20 40 60 80
Queue (front-->rear): 60 80
```

Методы `insert()` и `remove()` класса `LinkQueue` реализуются методами `insertLast()` и `deleteFirst()` класса `FirstLastList`. Массив, использованный для реализации очереди в программе `queue.java` (см. листинг 4.4) в главе 4, был заменен связанным списком.

Программы `linkStack.java` и `linkQueue.java` наглядно показывают, что стеки и очереди представляют собой концептуальные сущности, отделенные от их реализаций. Стек может быть с равным успехом реализован как на базе массива, так и на базе связанного списка. Для работы стека важны операции `push()`/`pop()` и то, как они используются, а не базовый механизм, выбранный для реализации этих операций.

Когда следует использовать реализацию стека или очереди на базе связанного списка, а когда — на базе массива? В частности, это зависит от того, насколько точно вы можете предсказать объем данных, которые будут храниться в стеке или очереди. Если объем данных неизвестен, связанный список обеспечивает большую гибкость, чем массив. Обе реализации работают достаточно быстро, поэтому скорость, вероятно, не будет определяющим фактором.

Типы данных и абстракция

Откуда взялся термин «абстрактные типы данных»? Давайте сначала разберемся с «типами данных» вообще, а затем вернемся к «абстрактности».

Типы данных

Выражение «тип данных» имеет достаточно широкий смысл. Когда-то оно применялось к встроенным типам данных — таким, как `int` и `double`. Обычно это первое, что приходит в голову при виде этого термина.

Термином «примитивный тип» в действительности обозначаются сразу два понятия: элемент данных, обладающий определенными характеристиками, и набор операций, разрешенных для этих данных. Например, переменные типа `int` в Java могут принимать целочисленные значения в диапазоне от –2 147 483 648 до +2 147 483 647, к которым могут применяться операторы +, –, *, / и т. д. Допустимые операции типа данных являются неотделимой частью его смыслового содержания; чтобы понять смысл типа, необходимо понять, какие операции могут с ним выполняться.

Объектно-ориентированное программирование позволяет программисту определять собственные типы данных в форме классов. Некоторые из этих типов представляют числовые величины, используемые по аналогии с примитивными типами. Например, можно определить класс для представления времени (с полями данных для хранения часов, минут, секунд), класс для представления дробей (с полями для хранения числителя и знаменателя) или класс для представления сверхдлинных чисел (строка, символы которой представляют цифры). Такие классы могут поддерживать операции сложения и вычитания, как и типы `int` и `double`, — разве что в Java вместо операторов + и — должны использоваться методы с функциональной записью вида `add()` и `sub()`.

Выражение «тип данных» естественным образом подходит для таких классов, ориентированных на работу с числовыми данными. Однако оно также распространяется и на классы, не обладающие этой характеристикой. Более того, любой класс представляет некоторый тип данных: он тоже состоит из данных (полей) и операций, разрешенных для этих данных (методов).

Если уж на то пошло, любая структура данных (стек, очередь и т. д.), представленная в виде класса, тоже может называться типом данных. Стек во многих отношениях отличается от `int`, но оба типа определяются как совокупность схемы данных и набора операций с этими данными.

Абстракция

Слово «абстрактный» означает «логически отделенный от подробного описания или реализации». Абстракция представляет сущность или важнейшие характеристики чего-либо. Например, «пост президента» — это абстракция, не зависящая от личности человека, занимающего этот пост. Конкретные лица, занимающие этот пост, приходят и уходят, а полномочия и обязанности поста президента остаются неизменными.

Таким образом, в объектно-ориентированном программировании абстрактным типом данных называется класс, рассматриваемый независимо от его реализации. Абстрактный тип складывается из описания данных класса (поля), списка выполняемых с этими данными операций (методы) и инструкций по выполнению операций. Из описания исключаются любые подробности того, как методы выполняют свои задачи. Пользователь класса знает, какие методы и как он должен вызывать, какие результаты он при этом может получить, но только не то, как методы выполняют свою работу.

Смысл понятия абстрактного типа данных дополнительно расширяется для структур данных — таких, как стеки и очереди. Как и для любого другого класса, этим термином обозначаются данные и выполняемые с ними операции, но в этом контексте даже сам способ хранения данных скрывается от пользователя. Пользователю неизвестно не только то, как метод выполняет свою работу, но и то, какая структура используется для хранения данных.

Скажем, применительно к стеку пользователь знает о существовании методов `push()` и `pop()` (и возможно, ряда других методов) и о том, как они работают. Пользователю (по крайней мере в общем случае) не нужно знать, как работают методы `push()` и `pop()`, а также хранятся ли данные в массиве, связанном списке или другой структуре данных — например, дереве.

Интерфейс

Спецификация абстрактного типа данных часто называется *интерфейсом*. В интерфейсе объединяется все, что видит пользователь класса — обычно его открытые методы. В классе стека интерфейс образуется из методов `push()`, `pop()` и других аналогичных методов.

Списки ADT

Теперь, когда вы знаете, что такое абстрактный тип данных, мы можем рассмотреть еще один пример: *список*. Список (иногда также называемый *линейным списком*) представляет собой группу элементов, расположенных друг за другом — как бусины на нити или дома на улице. Списки поддерживают некоторые фундаментальные операции; обычно в них можно вставлять элементы, удалять элементы, а также прочитать элемент в заданной позиции (скажем, третий элемент).

Не путайте списки ADT со связанными списками, которые рассматривались в этой главе. Список определяется своим интерфейсом, то есть конкретными мето-

дами, используемыми для взаимодействия с ним. Интерфейс может реализоваться на базе разных структур, в том числе массивов и связанных списков. Список является абстракцией таких структур данных.

Абстрактные типы данных как инструмент проектирования

Концепция абстрактных типов данных полезна в процессе проектирования архитектуры программного продукта. Если вам требуется хранить какие-либо данные, для начала выясните, какие операции должны выполняться с этими данными. Потребуется ли обращаться к последнему вставленному элементу? К первому? К элементу с заданным ключом? К элементу в заданной позиции? Ответы на такие вопросы приводят к определению абстрактного типа. Только после того, как абстрактный тип данных будет полностью определен, можно переходить к подробностям представления данных или программированию методов, работающих с этими данными.

Отделение спецификации ADT от подробностей реализации упрощает процесс проектирования. Кроме того, оно упростит возможные изменения реализации в будущем. Если пользователь имеет дело только с интерфейсом ADT, вы сможете изменять реализацию, не нарушая работоспособности готового пользовательского кода.

Конечно, после определения абстрактного типа необходимо тщательно выбрать базовые структуры данных, чтобы указанные операции выполнялись по возможности эффективно. Скажем, если вам необходим произвольный доступ к элементу N, представление в виде связанного списка вряд ли будет уместно, потому что операции произвольного доступа в связанных списках выполняются неэффективно. В данной ситуации лучше воспользоваться массивом.

ПРИМЕЧАНИЕ

Не забывайте, что концепция абстрактных типов данных является всего лишь концептуальным инструментом. Не стоит полагать, что структуры данных четко делятся на абстрактные и те, которые используются для их реализации. Например, связанный список полезен и без упаковки в интерфейс списка ADT; он может выполнять функции абстрактного типа сам по себе, а может использоваться для реализации другого типа данных (например, очереди). Возможна реализация связанного списка на базе массива, или наоборот — аналоги массива могут быть реализованы на базе связанного списка. Выбор между абстрактными типами данных и основными структурами зависит от контекста.

Сортированные списки

В связанных списках, встречавшихся нам до настоящего момента, данные хранились в произвольном порядке. Однако в некоторых приложениях бывает удобно хранить данные списка в упорядоченном виде. Список, обладающий этим свойством, называется *сортированным списком*.

В сортированном списке элементы сортируются по значению ключа. Удаление часто ограничивается наименьшим (или наибольшим) элементом, который

находится в начале списка, хотя иногда в работе с сортированными списками также применяются методы `find()` и `delete()`, которые ищут в списке элемент с заданным ключом.

В общем случае, сортированный список может использоваться почти во всех ситуациях, в которых может использоваться сортированный массив. Преимущества сортированного списка перед сортированным массивом — быстрая вставка (не требующая перемещения элементов) и возможность динамического расширения списка (тогда как массив ограничивается фиксированным размером). С другой стороны, сортированный список реализуется несколько сложнее, чем сортированный массив.

Позднее мы рассмотрим одно из практических применений сортированных списков: сортировку данных. Сортированные списки также могут использоваться для реализации приоритетных очередей, хотя чаще встречаются реализации на базе кучи (см. главу 12, «Пирамиды»).

Приложение LinkList Workshop, описанное в начале этой главы, демонстрирует работу как с сортированными, так и с несортированными списками. Чтобы увидеть, как работают сортированные списки, создайте новый список примерно из 20 элементов при помощи кнопки `New` и установите переключатель `Sorted`. В результате будет создан список с данными, следующими в порядке сортировки (рис. 5.11).

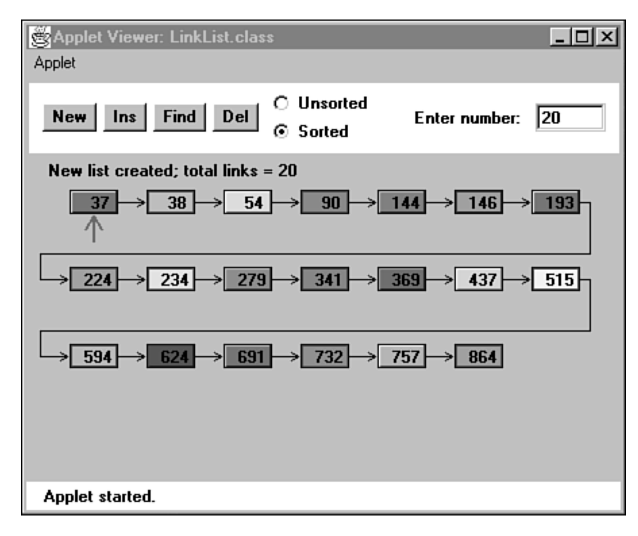


Рис. 5.11. Приложение LinkList Workshop с сортированным списком

Вставьте новый элемент при помощи кнопки `Ins`. Введите значение, находящееся примерно в середине списка. Проследите за тем, как алгоритм перебирает элементы в поисках места для вставки. Обнаружив подходящую позицию, он вставляет новый элемент (рис. 5.12).

При следующем нажатии `Ins` происходит перерисовка списка для придания ему нормального вида. Также приложение позволяет найти заданный элемент при помощи кнопки `Find` и удалить заданный элемент при помощи кнопки `Del`.

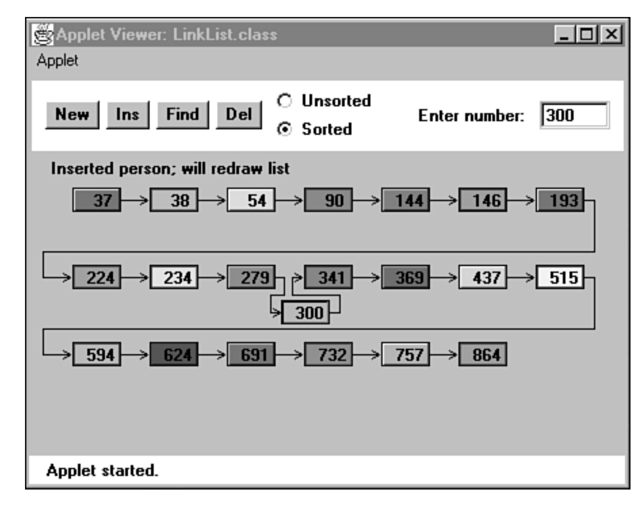


Рис. 5.12. Вставка нового элемента

Реализация вставки элемента в сортированный список на языке Java

Чтобы вставить элемент в отсортированный список, алгоритм сначала перебирает элементы в поисках подходящей позиции для вставки (то есть позиции перед первым элементом, большим вставляемого, рис. 5.12).

Обнаружив позицию для вставки, алгоритм вставляет элемент обычным способом: в поле next нового элемента заносится ссылка на следующий элемент, а в поле next предыдущего элемента заносится ссылка на новый элемент. Однако при этом приходится учитывать особые случаи: вставку нового элемента в начале или в конце списка. Код метода вставки:

```
public void insert(long key) // Вставка в порядке сортировки
   {
   Link newLink = new Link(key);    // Создание нового элемента
   Link previous = null;            // От начала списка
   Link current = first;
                                    // До конца списка
   while(current != null && key > current.dData)
      {                             // или если key > current,
      previous = current;
      current = current.next;       // Перейти к следующему элементу
      }
   if(previous==null)               // В начале списка
      first = newLink;              // first --> newLink
   else                             // Не в начале
      previous.next = newLink;      // старое значение prev --> newLink
   newLink.next = current;          // newLink --> старое значение current
   }
```

При перемещении сохраняется ссылка на предыдущий элемент, чтобы в поле next предыдущего элемента можно было записать ссылку на новый элемент. После создания нового элемента метод готовится к поиску: для этого current, как обычно, инициализируется значением first. Переменной previous также присваивается null; эта операция важна, потому что позднее по этому значению метод будет проверять, находится ли текущая позиция в начале списка.

Цикл while не отличается от тех, которые использовались ранее для поиска позиции вставки, но в нем появилось новое условие. Цикл прерывается в том случае, если ключ текущего элемента (current.dData) не меньше ключа вставляемого элемента (key); это самый типичный случай, когда элемент вставляется где-то в середине списка.

Однако цикл while также прерывается и в том случае, если переменная current содержит null. Это происходит в конце списка (поле next последнего элемента равно null), а также в том случае, если список изначально пуст (first содержит null). Таким образом, при завершении цикла while текущая позиция может находиться в начале, в середине или в конце списка; кроме того, список может быть пустым.

Если текущая позиция находится в начале или список пуст, переменная previous будет равна null; соответственно в поле first заносится ссылка на новый элемент. В противном случае текущая позиция находится в середине или в конце списка, и в поле previous.next заносится ссылка на новый элемент.

В любом случае полю next нового элемента присваивается current. Если текущая позиция находится в конце списка, то поле current равно null, так что полю next нового элемента будет правильно присвоено именно это значение.

Программа sortedList.java

В программе sortedList.java (листинг 5.6) представлен класс SortedList с методами insert(), remove() и displayList(). Только метод insert() отличается от своего аналога из класса несортированного списка.

Листинг 5.6. Программа sortedList.java

```
// sortedList.java
// Работа с сортированным списком
// Запуск программы: C>java SortedListApp
////////////////////////////////////////////////////////////////
class Link
   {
   public long dData;                  // Данные
   public Link next;                   // Ссылка на следующий элемент списка
// -------------------------------------------------------------
   public Link(long dd)                // Конструктор
      { dData = dd; }
// -------------------------------------------------------------
   public void displayLink()           // Вывод содержимого элемента
      { System.out.print(dData + " "); }
}  // end class Link
```

```
////////////////////////////////////////////////////////////////
class SortedList
   {
   private Link first;                 // Ссылка на первый элемент списка
// -------------------------------------------------------------
   public SortedList()                 // Конструктор
      { first = null; }
// -------------------------------------------------------------
   public boolean isEmpty()            // true, если список пуст
      { return (first==null); }
// -------------------------------------------------------------
   public void insert(long key)        // Вставка в порядке сортировки
      {
      Link newLink = new Link(key);    // Создание нового элемента
      Link previous = null;            // От начала списка
      Link current = first;
                                       // До конца списка
      while(current != null && key > current.dData)
         {                             // или если key > current,
         previous = current;
         current = current.next;       // Перейти к следующему элементу
         }
      if(previous==null)               // В начале списка
         first = newLink;              // first --> newLink
      else                             // Не в начале
         previous.next = newLink;      // старое значение prev --> newLink
      newLink.next = current;        // newLink --> старое значение current
   }
// -------------------------------------------------------------
   public Link remove()           // Удаление первого элемента
      {                           // (предполагается, что список не пуст)
      Link temp = first;          // Сохранение ссылки
      first = first.next;      // Удаление: first-->ссылка на второй элемент
      return temp;                // Метод возвращает ссылку
      }                           // на удаленный элемент
   }
// -------------------------------------------------------------
   public void displayList()
      {
         System.out.print("List (first-->last): ");
         Link current = first;       // От начала списка
         while(current != null)      // Перемещение до конца списка
            {
            current.displayLink();   // Вывод данных
            current = current.next;  // Переход к следующему элементу
            }
            System.out.println("");
      }
   }  // Конец класса SortedList
////////////////////////////////////////////////////////////////
```

продолжение ⇨

Листинг 5.6 (*продолжение*)

```
class SortedListApp
   {
   public static void main(String[] args)
      {                             // Создание нового списка
      SortedList theSortedList = new SortedList();
      theSortedList.insert(20);    // Вставка двух элементов
      theSortedList.insert(40);

      theSortedList.displayList(); // Вывод содержимого списка

      theSortedList.insert(10);    // Вставка трех элементов
      theSortedList.insert(30);
      theSortedList.insert(50);

      theSortedList.displayList(); // Вывод содержимого списка

      theSortedList.remove();      // Удаление элемента

      theSortedList.displayList(); // Вывод содержимого списка
      }
   }  // Конец класса SortedListApp
////////////////////////////////////////////////////////////////
```

Метод `main()` вставляет в список два элемента с ключами 20 и 40, после чего вставляются еще три элемента с ключами 10, 30 и 50. Элементы вставляются как в начале списка, так и в середине и в конце — это показывает, что метод `insert()` правильно обрабатывает все эти особые случаи. В завершение один элемент удаляется из списка, демонстрируя тем самым, что удаление всегда выполняется от начала списка. После каждого изменения выводится текущее содержимое списка. Результат выполнения программы `sortedList.java`:

```
List (first-->last): 20 40
List (first-->last): 10 20 30 40 50
List (first-->last): 20 30 40 50
```

Эффективность сортированных списков

Вставка и удаление произвольных элементов в сортированных связанных списках требуют $O(N)$ сравнений (в среднем $N/2$), потому что позицию для выполнения операции приходится искать перебором списка. С другой стороны, поиск или удаление наименьшего значения выполняется за время $O(1)$, потому что оно всегда находится в начале списка. Если приложение часто обращается к наименьшему элементу, а скорость вставки не критична, то сортированный связанный список будет достаточно эффективным. Например, приоритетная очередь может быть реализована на базе сортированного связанного списка.

Сортировка методом вставки

Сортированный список может использоваться в качестве достаточно эффективного механизма сортировки. Допустим, у вас имеется массив с несортированными данными. Если последовательно читать элементы из массива и вставлять их в сортированный список, они будут автоматически отсортированы. Остается извлечь их из списка и поместить обратно в массив — они будут отсортированы.

Подобный вид сортировки заметно превосходит по эффективности обычные виды сортировки методом вставки в массиве (см. главу 3, «Простая сортировка»), потому что она требует меньшего количества операций копирования. Она также выполняется за время $O(N^2)$, потому что при вставке в сортированный список каждый элемент приходится сравнивать в среднем с половиной элементов, уже находящихся в списке; для N вставляемых элементов количество сравнений составит $N^2/4$. Однако каждый элемент в этом случае копируется только два раза: из массива в список и из списка в массив. $N \times 2$ операций копирования выгодно отличается от сортировки методом вставки в массив, требующей в среднем N^2 таких операций.

В листинге 5.7 приведена программа `listInsertionSort.java`, которая создает несортированный массив элементов типа `link`, вставляет их в сортированный список (с использованием конструктора), после чего удаляет и помещает обратно в массив.

Листинг 5.7. Программа listInsertionSort.java

```
// listInsertionSort.java
// Применение сортированного списка для сортировки массива
// Запуск программы: C>java ListInsertionSortApp
////////////////////////////////////////////////////////////////
class Link
   {
   public long dData;                 // Данные
   public Link next;                  // Следующий элемент списка
// -------------------------------------------------------------
   public Link(long dd)               // Конструктор
      { dData = dd; }
// -------------------------------------------------------------
   }  // Конец класса Link
////////////////////////////////////////////////////////////////
class SortedList
   {
   private Link first;           // Ссылка на первый элемент списка
// -------------------------------------------------------------
   public SortedList()           // Конструктор (без аргументов)
      { first = null; }          // Инициализация списка
// -------------------------------------------------------------
   public SortedList(Link[] linkArr)  // Конструктор (аргумент - массив)
      {                               //
      first = null;                        // Инициализация списка
      for(int j=0; j<linkArr.length; j++)  // Копирование массива
```

продолжение ⇗

Листинг 5.7 (*продолжение*)

```
        insert( linkArr[j] );             // в список
      }
// -------------------------------------------------------------
   public void insert(Link k)     // Вставка (в порядке сортировки)
      {
      Link previous = null;            // От начала списка
      Link current = first;
                                       // До конца списка
      while(current != null && k.dData > current.dData)
         {                             // или если ключ > current,
         previous = current;
         current = current.next;       // Перейти к следующему элементу
         }
      if(previous==null)               // В начале списка
         first = k;                    // first --> k
      else                             // Не в начале
         previous.next = k;            // старое значение prev --> k
      k.next = current;                // k --> старое значение current
      }
// -------------------------------------------------------------
   public Link remove()           // Возвращает и удаляет первую ссылку
      {                           // (assumes non-empty list)
      Link temp = first;               // Сохранение ссылки
      first = first.next;              // Удаление первого элемента
      return temp;                     // Метод возвращает ссылку
      }                                // на удаленный элемент
// -------------------------------------------------------------
   }  // Конец класса SortedList

////////////////////////////////////////////////////////////////
class ListInsertionSortApp
   {
   public static void main(String[] args)
      {
      int size = 10;
                                        // Создание массива
      Link[] linkArray = new Link[size];

      for(int j=0; j<size; j++)        // Заполнение массива
         {                             // Случайные числа
         int n = (int)(java.lang.Math.random()*99);
         Link newLink = new Link(n);   // Создание элемента
         linkArray[j] = newLink;       // Сохранение в массиве
         }
                                       // Вывод содержимого массива
         System.out.print("Unsorted array: ");
         for(int j=0; j<size; j++)
            System.out.print( linkArray[j].dData + " " );
```

```
      System.out.println("");
                    // Создание нового списка,
                    // инициализированного содержимым массива
      SortedList theSortedList = new SortedList(linkArray);

      for(int j=0; j<size; j++)  // links from list to array
         linkArray[j] = theSortedList.remove();
                              // Вывод содержимого массива
      System.out.print("Sorted Array:   ");
      for(int j=0; j<size; j++)
         System.out.print(linkArray[j].dData + " ");
      System.out.println("");
      }
   }  // Конец класса ListInsertionSortApp
////////////////////////////////////////////////////////////////
```

Программа выводит элементы массива до и после операции сортировки. Пример вывода:

```
Unsorted array: 59 69 41 56 84 15 86 81 37 35
Sorted array:   15 35 37 41 56 59 69 81 84 86
```

При каждом запуске выводится новый результат, потому что исходные элементы генерируются случайным образом.

Новый конструктор `SortedList` получает в аргументе массив объектов `Link` и вставляет все содержимое массива в созданный список. Это упрощает работу со списком для клиента (метода `main()`).

Изменения также внесены в метод `insert()`. В новой версии метод получает аргумент типа `Link` вместо `long`. Объекты `Link` сохраняются в массиве и вставляются непосредственно в список. В программе `sortedList.java` (см. листинг 5.6) было удобнее, чтобы метод `insert()` создавал каждый объект `Link` по значению `long`, переданному в аргументе.

Недостаток сортировки методом вставки в список (по сравнению с сортировкой методом вставки в массив) заключается в том, что она занимает вдвое больше памяти, так как массив и связанный список должны находиться в памяти одновременно. И все же, если у вас имеется готовый класс сортированного списка, такой способ сортировки хорошо подходит для не очень больших массивов.

Двусвязные списки

Рассмотрим следующую разновидность связанных списков: *двусвязный* список (не путайте с двусторонним списком!). Потенциальным недостатком обычных связанных списков является сложность перемещения по списку в обратном направлении. Команда вида

```
current=current.next
```

позволяет легко перейти к следующему элементу списка, но соответствующего способа перехода к предыдущему элементу не существует. В некоторых ситуациях это

ограничение создает проблемы. Допустим, в текстовом редакторе связанный список используется для хранения текста. Каждая строка на экране хранится в виде объекта String, встроенного в элемент списка. Когда пользователь перемещает курсор вниз по экрану, программа переходит к следующему элементу списка для обработки или вывода следующей строки. Но что произойдет, если пользователь переместит курсор вверх? При использовании обычного связанного списка придется вернуть переменную current (или ее аналог) к началу списка, а затем перебрать все элементы до предыдущего элемента. Конечно, такой способ неэффективен — перемещение к предыдущей строке должно осуществляться за один шаг.

Двусвязный список предоставляет такую возможность. Он позволяет перемещаться по списку как в прямом, так и в обратном направлении. Дело в том, что каждый элемент хранит ссылки на два других элемента вместо одного. Первая ссылка указывает на следующий элемент, как и в обычных списках. Вторая ссылка указывает на предыдущий элемент. Структура такого списка изображена на рис. 5.13.

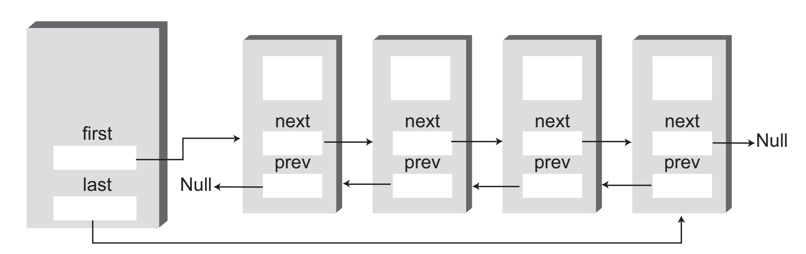


Рис. 5.13. Двусвязный список

Начало определения класса Link, представляющего элемент двусвязного списка, выглядит примерно так:

```
class Link
   {
   public long dData;                 // Данные
   public Link next;                  // Ссылка на следующий элемент списка
   public link previous;              // Ссылка на предыдущий элемент списка
...
}
```

К недостаткам двусвязных списков следует отнести то, что при каждой вставке или удалении ссылки вам приходится изменять четыре ссылки вместо двух: две связи с предыдущим элементом и две связи со следующим элементом. И конечно, каждый элемент списка занимает чуть больше места из-за дополнительной ссылки.

Двусвязный список не обязан быть двусторонним (то есть ссылка на последний элемент не обязана храниться в объекте списка), но такая реализация удобна, поэтому мы включим ее в наш пример.

Полный код программы doublyLinked.java будет приведен ниже, а пока рассмотрим подробнее некоторые методы класса doublyLinkedList.

Перебор

Два метода, имена которых начинаются с display, демонстрируют перебор в двусвязном списке. Метод displayForward() не отличается от метода displayList() обычного связанного списка. Метод displayBackward() похож на него, но он начинает перебор с последнего элемента и перемещается к началу списка по значению поля previous каждого элемента. Следующий фрагмент кода показывает, как работает этот процесс:

```
Link current = last;           // От начала списка
while(current != null)         // Перемещение до конца списка
   current = current.previous; // Переход к предыдущему элементу
```

Вставка

Класс DoublyLinkedList содержит несколько методов вставки. Метод insertFirst() вставляет новый элемент в начале списка, метод insertLast() — в конце, а метод insertAfter() — после элемента с заданным ключом.

Убедившись в том, что список не пуст, метод insertFirst() записывает ссылку на новый элемент в поле previous «старого» первого элемента, а ссылку на «старый» первый элемент — в поле next нового элемента. Наконец, в поле first заносится ссылка на новый элемент. Процесс изображен на рис. 5.14.

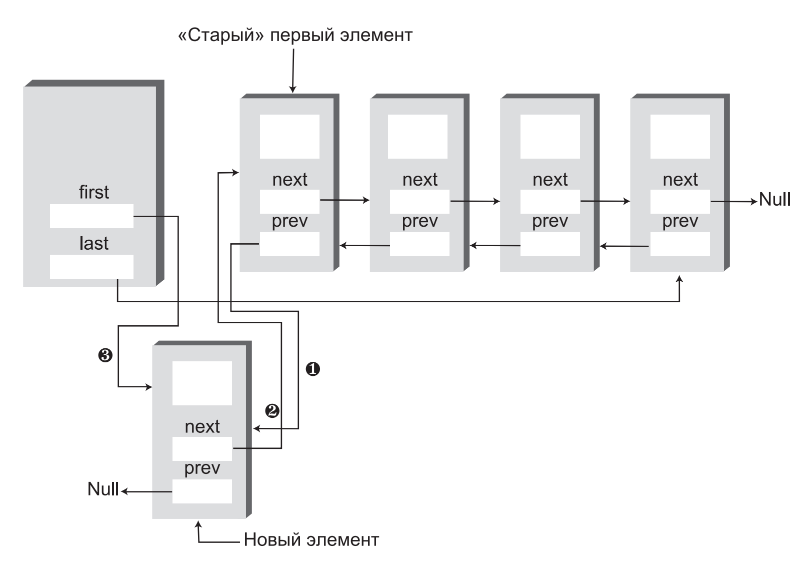


Рис. 5.14. Вставка в начале

Если список пуст, то вместо поля first.previous необходимо изменить поле last. Вот как это делается:

```
if( isEmpty() )              // Если список не содержит элементов,
   last = newLink;           // newLink <-- last
```

```
else
   first.previous = newLink;   // newLink <-- старое значение first
newLink.next = first;          // newLink --> старое значение first
first = newLink;               // first --> newLink
```

Метод insertLast() выполняет те же действия в конце списка; он является своего рода «зеркальным отражением» insertFirst().

Метод insertAfter() вставляет новый элемент после элемента с заданным ключом. Операция несколько усложняется, потому что в этой ситуации необходимо изменить четыре ссылки. Прежде всего следует найти элемент с заданным ключом; поиск выполняется так же, как в методе find() программы linkList2.java (см. листинг 5.2). Затем, если позиция вставки находится не в конце списка, необходимо создать две связи между новым и следующим элементом, и еще две — между current и новым элементом. Процесс показан на рис. 5.15.

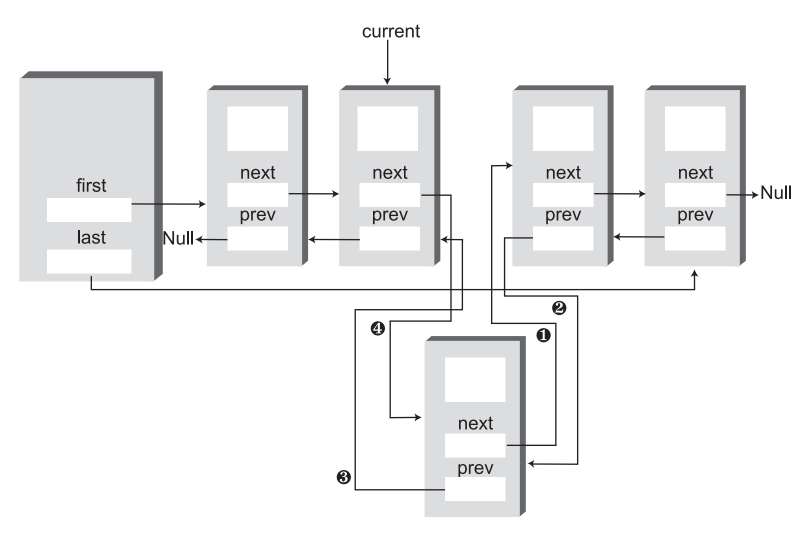


Рис. 5.15. Вставка в произвольной позиции

Если новый элемент должен вставляться в конце списка, то его поле next должно содержать null, а поле last — ссылку на новый элемент. Фрагмент insertAfter(), в котором настраиваются ссылки, выглядит так:

```
if(current==last)              // Для последнего элемента списка
   {
   newLink.next = null;        // newLink --> null
   last = newLink;             // newLink <-- last
   }
else                           // Не последний элемент
   {
   newLink.next = current.next; // newLink --> старое значение next
                                // newLink <-- старое значение next
   current.next.previous = newLink;
   }
```

```
newLink.previous = current;    // старое значение current <-- newLink
current.next = newLink;        // старое значение current --> newLink
```

Возможно, вам еще не встречались выражения с двумя операторами «точка»? Это естественное расширение стандартного оператора «точка». Выражение

```
current.next.previous
```

обозначает поле `previous` элемента, на который ссылается поле `next` элемента `current`.

Удаление

Класс также содержит три метода удаления: `deleteFirst()`, `deleteLast()` и `deleteKey()`. Первые два относительно тривиальны; в методе `deleteKey()` удаляется элемент `current`. Если удаляемый элемент не является ни первым, ни последним в списке, то в поле `next` элемента `current.previous` (элемент, предшествующий удаляемому) заносится ссылка на `current.next` (элемент, следующий после удаляемого), а в поле `previous` элемента `current.next` заносится ссылка на `current.previous`. В результате элемент `current` исключается из списка. На рис. 5.16 показано, как происходит удаление, реализованное следующими двумя командами:

```
current.previous.next = current.next;
current.next.previous = current.previous;
```

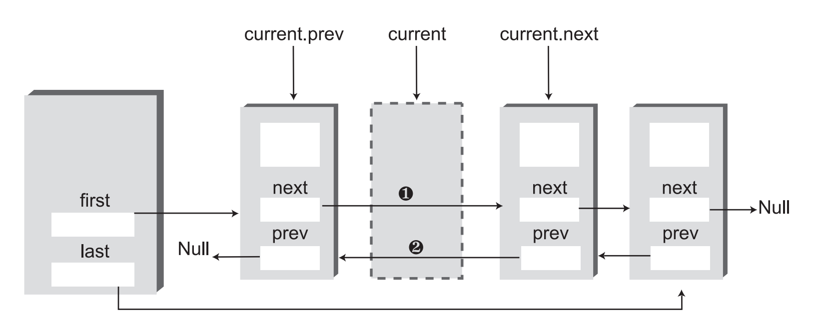


Рис. 5.16. Удаление произвольного элемента

Если удаляемый элемент находится в первой или последней позиции списка, это особый случай, потому что ссылка на следующий или предыдущий элемент должна быть сохранена в поле `first` или `last`. Приведем соответствующий фрагмент кода `deleteKey()` для изменения ссылок:

```
if(current==first)             // Первый элемент?
   first = current.next;       // first --> старое значение next
else                           // Не первый элемент
   // старое значение previous --> старое значение next
   current.previous.next = current.next;

if(current==last)              // Последний элемент?
   last = current.previous;    // старое значение previous <-- last
```

```
else                          // Не последний элемент
   // старое значение previous <-- старое значение next
   current.next.previous = current.previous;
```

Программа doublyLinked.java

В листинге 5.8 приведен полный код программы `doublyLinked.java` вместе со всеми упоминавшимися методами.

Листинг 5.8. Программа doublyLinked.java

```
// doublyLinked.java
// Двусвязный список
// Запуск программы: C>java DoublyLinkedApp
////////////////////////////////////////////////////////////////
class Link
   {
   public long dData;                 // Данные
   public Link next;                  // Следующий элемент в списке
   public Link previous;              // Предыдущий элемент в списке
// -------------------------------------------------------------
   public Link(long d)                // Конструктор
      { dData = d; }
// -------------------------------------------------------------
   public void displayLink()          // Вывод содержимого элемента
      { System.out.print(dData + " "); }
// -------------------------------------------------------------
   }  // Конец класса Link
////////////////////////////////////////////////////////////////
class DoublyLinkedList
   {
   private Link first;               // Ссылка на первый элемент списка
   private Link last;                // Ссылка на последний элемент списка
// -------------------------------------------------------------
   public DoublyLinkedList()         // Конструктор
      {
      first = null;                  // Список пока не содержит элементов
      last = null;
      }
// -------------------------------------------------------------
   public boolean isEmpty()          // true, если список пуст
      { return first==null; }
// -------------------------------------------------------------
   public void insertFirst(long dd)  // Вставка элемента в начало списка
      {
      Link newLink = new Link(dd);   // Создание нового элемента
      if( isEmpty() )                // Если список не содержит элементов,
         last = newLink;             // newLink <-- last
      else
         first.previous = newLink;   // newLink <-- старое значение first
```

```
      newLink.next = first;          // newLink --> старое значение first
      first = newLink;               // first --> newLink
   }
// -------------------------------------------------------------
   public void insertLast(long dd)   // элемент в конец списка
      {
      Link newLink = new Link(dd);   // Создание нового элемента
      if( isEmpty() )                // Если список не содержит элементов,
         first = newLink;            // first --> newLink
      else
         {
         last.next = newLink;        // старое значение last --> newLink
         newLink.previous = last;    // старое значение last <-- newLink
         }
      last = newLink;                // newLink <-- last
   }
// -------------------------------------------------------------
   public Link deleteFirst()         // Удаление первого элемента
      {                              // (предполагается, что список не пуст)
      Link temp = first;
      if(first.next == null)         // Если только один элемент
         last = null;                // null <-- last
      else
         first.next.previous = null; // null <-- старое значение next
      first = first.next;            // first --> старое значение next
      return temp;
      }
// -------------------------------------------------------------
   public Link deleteLast()          // Удаление последнего элемента
      {                              // (предполагается, что список не пуст)
      Link temp = last;
      if(first.next == null)         // Если только один элемент
         first = null;               // first --> null
      else
         last.previous.next = null;  // старое значение previous --> null
      last = last.previous;          // старое значение previous <-- last
      return temp;
}
// -------------------------------------------------------------
   // Вставка dd в позицию после key
   public boolean insertAfter(long key, long dd)
      {                              // (предполагается, что список не пуст)
      Link current = first;          // От начала списка
      while(current.dData != key)    // Пока не будет найдено совпадение
         {
         current = current.next;     // Переход к следующему элементу
         if(current == null)
            return false;            // Ключ не найден
         }
```

продолжение ⇨

Листинг 5.7 (*продолжение*)

```
      Link newLink = new Link(dd);   // Создание нового элемента

      if(current==last)              // Для последнего элемента списка
         {
         newLink.next = null;        // newLink --> null
         last = newLink;             // newLink <-- last
         }
      else                           // Не последний элемент
         {
         newLink.next = current.next; // newLink --> старое значение next
                                      // newLink <-- старое значение next
         current.next.previous = newLink;
         }
      newLink.previous = current;    // старое значение current <-- newLink
      current.next = newLink;        // старое значение current --> newLink
      return true;                   // Ключ найден, вставка выполнена
      }
// -------------------------------------------------------------
   public Link deleteKey(long key)   // Удаление элемента с заданным ключом
      {                              // (предполагается, что список не пуст)
      Link current = first;          // От начала списка
      while(current.dData != key)    // Пока не будет найдено совпадение
         {
         current = current.next;     // Переход к следующему элементу
         if(current == null)
            return null;             // Ключ не найден
         }
      if(current==first)             // Ключ найден; это первый элемент?
         first = current.next;       // first --> старое значение next
      else                           // Не первый элемент
         // старое значение previous --> старое значение next
         current.previous.next = current.next;

      if(current==last)              // Последний элемент?
         last = current.previous;    // старое значение previous <-- last
      else                           // Не последний элемент
         // Старое значение previous <-- старое значение next
         current.next.previous = current.previous;
      return current;                // Возвращение удаленного элемента
      }
// -------------------------------------------------------------
   public void displayForward()
      {
      System.out.print("List (first-->last): ");
      Link current = first;          // От начала списка
      while(current != null)         // Перемещение до конца списка
         {
         current.displayLink();      // Вывод данных
```

```
         current = current.next;     // Переход к следующему элементу
         }
      System.out.println("");
      }
// -------------------------------------------------------------
   public void displayBackward()
      {
      System.out.print("List (last-->first): ");
      Link current = last;           // От начала списка
      while(current != null)         // Перемещение до конца списка
         {
         current.displayLink();      // Вывод данных
         current = current.previous; // Переход к следующему элементу
         }
      System.out.println("");
      }
// -------------------------------------------------------------
   }  // Конец класса DoublyLinkedList
////////////////////////////////////////////////////////////////
class DoublyLinkedApp
   {
   public static void main(String[] args)
      {                             // Создание нового списка
      DoublyLinkedList theList = new DoublyLinkedList();

      theList.insertFirst(22);      // Вставка в начале
      theList.insertFirst(44);
      theList.insertFirst(66);

      theList.insertLast(11);       // Вставка в конце
      theList.insertLast(33);
      theList.insertLast(55);

      theList.displayForward();     // Вывод в прямом направлении
      theList.displayBackward();    // Вывод в обратном направлении

      theList.deleteFirst();        // Удаление первого элемента
      theList.deleteLast();         // Удаление последнего элемента
      theList.deleteKey(11);        // Удаление элемента с ключом 11

      theList.displayForward();     // Вывод в прямом направлении

      theList.insertAfter(22, 77);  // Вставка 77 после 22
      theList.insertAfter(33, 88);  // Вставка 88 после 33

      theList.displayForward();     // Вывод в прямом направлении
      }
   }  // Конец класса DoublyLinkedApp
////////////////////////////////////////////////////////////////
```

Метод main() вставляет элементы в начале и в конце списка, выводит содержимое списка в прямом и обратном направлении и затем удаляет первый и последний элементы, а также элемент с ключом 11. Далее содержимое списка выводится снова (только в прямом направлении), в список вставляются два элемента с использованием метода insertAfter(), после чего список выводится снова. Результат выполнения программы:

```
List (first-->last): 66 44 22 11 33 55
List (last-->first): 55 33 11 22 44 66
List (first-->last): 44 22 33
List (first-->last): 44 22 77 33 88
```

Методы удаления и insertAfter() предполагают, что список не пуст. В реальном коде следовало бы включить в метод main() вызов isEmpty(), проверяющий наличие элементов в списке перед операциями вставки/удаления.

Двусвязный список как база для построения дека

Двусвязный список может использоваться в качестве базы для построения дека, или двусторонней очереди (см. предыдущую главу). В деке операции вставки и удаления могут выполняться с обоих концов; двусвязный список предоставляет такую возможность.

Итераторы

Вы уже видели, как происходит поиск элемента с заданным ключом с помощью метода find(). Метод начинает с первого элемента и последовательно перебирает все элементы до тех пор, пока не найдет нужный элемент. Другие рассмотренные операции — удаление элемента с заданным ключом, вставка до или после заданного элемента — тоже требуют перебора списка с целью нахождения элемента, с которым должна выполняться операция. Однако во всех этих методах пользователь не может как-либо управлять процессом перебора элементов.

Допустим, требуется перебрать элементы списка, выполняя некие операции с определенными элементами — скажем, увеличить зарплату всех работников, которые получают минимальный оклад (а записи других работников должны остаться без изменений). Или, скажем, из списка клиентов интернет-магазина удаляются записи всех покупателей, которые ничего не заказывали за последние полгода.

С массивами подобные операции выполняются просто, потому что для отслеживания текущей позиции можно воспользоваться индексом. Вы выполняете операцию с элементом, затем увеличиваете индекс, обращаетесь к следующему элементу и проверяете, является ли он подходящим кандидатом для выполнения операции. Однако в связанном списке элементы не имеют индексов. Как предоставить в распоряжение пользователя некий аналог индекса? Многократный вызов find() для поиска подходящих элементов списка потребует слишком большого количества сравнений для каждого элемента. Намного эффективнее было бы переходить от

элемента к элементу, проверять, удовлетворяет ли элемент некоторому критерию, и если проверка дает положительный результат — выполнять с ним соответствующую операцию.

Ссылка на элемент списка?

Фактически пользователю класса требуется ссылка, которая может указывать на произвольный элемент списка. По ссылке пользователь может проверить или изменить элемент списка. При этом должен поддерживаться механизм последовательного увеличения ссылки, чтобы каждое новое значение ссылки указывало на очередной элемент списка.

Допустим, мы создали такую ссылку; где она должна размещаться? Один из возможных вариантов — включить соответствующее поле непосредственно в список (оно может называться current или как-нибудь в этом роде). Пользователь обращается к элементу по ссылке current, а затем увеличивает current для перехода к следующему элементу.

Однако у такого подхода имеется серьезный недостаток: пользователю может потребоваться сразу несколько таких ссылок (по аналогии с одновременным использованием нескольких индексов при работе с массивом). Невозможно заранее предсказать, сколько полей может понадобиться. Соответственно будет проще предоставить пользователю возможность создать столько ссылок, сколько ему необходимо. Наиболее естественным решением для объектно-ориентированного языка будет внедрение каждой ссылки в объект класса. Это не может быть класс списка, так как объект списка существует в одном экземпляре, поэтому обычно ссылка реализуется в отдельном классе.

Итератор

Объекты, содержащие ссылки на элементы структур данных и используемые для перебора элементов этих структур, обычно называются *итераторами*. Определение класса итератора выглядит примерно так:

```
class ListIterator()
   {
   private Link current;
   ...
   }
```

Поле current содержит ссылку на элемент, на который в настоящее время указывает итератор. (Термин «указывает» в данном случае не имеет никакого отношения к указателям C++; мы используем его в более общем смысле «ссылается».)

Чтобы использовать итератор, пользователь сначала создает список, а затем объект-итератор, ассоциированный с этим списком. На самом деле проще поручить создание итератора списку, потому что он может передать итератору полезную информацию — скажем, ссылку на first. По этой причине в класс списка включается метод getIterator(), который возвращает пользователю объект-итератор для

данного списка. Сокращенный фрагмент кода `main()`, демонстрирующий основные принципы работы с итератором:

```
public static void main(...)
   {
   LinkList theList = new LinkList();          // Создание списка
   ListIterator iter1 = theList.getIterator();  // Создание итератора
   Link aLink = iter1.getCurrent();  // Обращение к элементу через итератор
   iter1.nextLink();      // Перемещение итератора к следующему элементу
   }
```

Созданный объект-итератор используется для обращения к элементу, на который он указывает, или увеличивается для перехода к следующему элементу (две последние команды). Использованное в этом фрагменте имя объекта-итератора `iter1` подчеркивает, что аналогичным образом можно создать и другие итераторы (`iter2` и т. д.).

Итератор всегда указывает на некоторый элемент списка. Он связан со списком, но не относится ни к классу списка, ни к классу элемента списка. На рис. 5.17 изображены два итератора, указывающих на разные элементы списка.

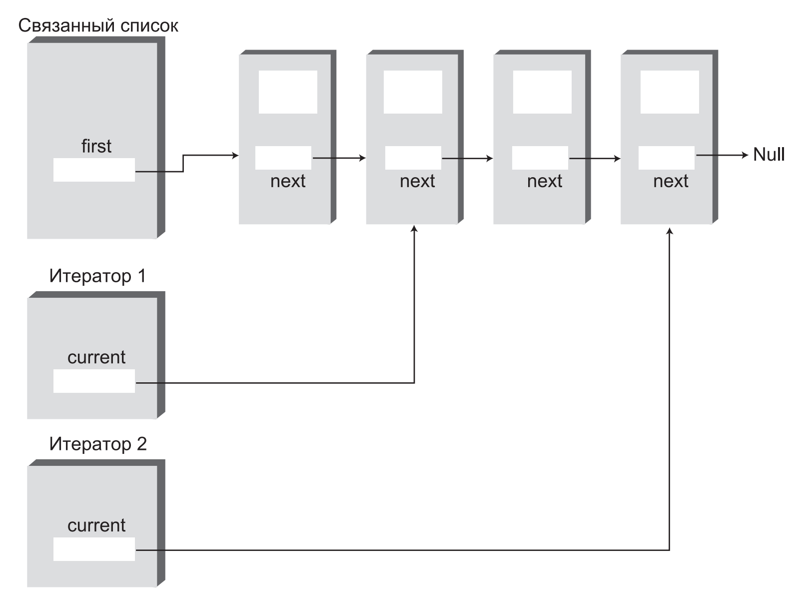


Рис. 5.17. Список и итераторы

Другие возможности итераторов

Мы уже рассмотрели несколько примеров того, как использование поля `previous` упрощало выполнение некоторых операций — скажем, удаление элемента в произвольной позиции. Такое поле также может оказаться полезным и для итератора.

Кроме того, итератору может потребоваться изменить значение поля `first` объекта списка — например, при вставке или удалении элемента в начале списка.

Если итератор является объектом отдельного класса, как он будет обращаться к приватному полю списка (каковым является `first`)? В одном из решений список передает итератору ссылку «на самого себя» при создании итератора. Ссылка сохраняется в поле итератора. В этом случае список должен предоставить открытые методы, при помощи которых итератор сможет изменить `first`. Эти методы `LinkList` называются `getFirst()` и `setFirst()`. (Недостаток такого решения заключается в том, что эти методы позволят изменить `first` любому желающему, что создает определенный риск.)

Ниже приведена переработанная (хотя все еще незавершенная) версия класса итератора, в которую включены эти дополнительные поля вместе с методами `reset()` и `nextLink()`:

```
class ListIterator()
   {
   private Link current;       // Ссылка на текущий элемент списка
   private Link previous;      // Ссылка на предыдущий элемент списка
   private LinkList ourList;   // Ссылка на "родительский" список
   public void reset()         // set to start of list
      {
      current = ourList.getFirst();  // current --> first
      previous = null;               // previous --> null
      }
   public void nextLink()      // Переход к следующему элементу
      {
      previous = current;      // Присваивание текущего элемента previous
      current = current.next;  // Присваивание текущему элементу next
      }
   ...
   }
```

Специально для опытных программистов C++: в C++ связь между итератором и списком обычно устанавливалась посредством объявления класса итератора *дружественным* (friend) по отношению к классу списка. В Java дружественных классов нет; впрочем, их полезность в любом случае спорна, поскольку дружественные классы создают трещинку в броне защиты данных.

Методы итераторов

Дополнительные методы наделяют класс итератора гибкостью и мощью. Все операции, связанные с перебором элементов списка (такие, как `insertAfter()`), которые ранее выполнялись классом списка, более естественно выполняются итератором. В нашем примере итератор включает следующие методы:

- `reset()` — перемещение итератора в начало списка.
- `nextLink()` — перемещение итератора к следующему элементу.
- `getCurrent()` — получение элемента, на который указывает итератор.
- `atEnd()` — true, если итератор находится в конце списка.
- `insertAfter()` — вставка нового элемента после итератора.

- insertBefore() — вставка нового элемента перед итератором.
- deleteCurrent() — удаление элемента в текущей позиции итератора.

Пользователь изменяет текущую позицию итератора методами reset() и nextLink(), проверяет достижение конца списка методом atEnd(), а также выполняет другие операции.

Решить, какие операции должны выполняться итератором, а какие — самим списком, бывает непросто. Метод insertBefore() лучше всего работает в итераторе, а метод insertFirst(), который всегда выполняет вставку в начале списка, уместнее разместить в классе списка. Мы оставили метод displayList() в списке, но эта операция может выполняться с использованием методов итератора getCurrent() и nextLink().

Программа interIterator.java

Интерактивный интерфейс программы interIterator.java дает возможность пользователю напрямую управлять итератором. После запуска программы операции выполняются нажатием соответствующей клавиши:

- s — вывод содержимого списка.
- r — возвращение итератора в начало списка.
- n — переход к следующему элементу.
- g — вывод содержимого текущего элемента.
- b — вставка перед текущим элементом.
- a — вставка нового элемента после текущего.
- d — удаление текущего элемента.

В листинге 5.9 приведен полный код программы interIterator.java.

Листинг 5.9

```
// interIterator.java
// Использование итератора связанного списка
// Запуск программы: C>java InterIterApp
import java.io.*;                 // Для ввода/вывода
////////////////////////////////////////////////////////////////
class Link
   {
   public long dData;             // Данные
   public Link next;              // Следующий элемент в списке
// -------------------------------------------------------------
   public Link(long dd)           // Конструктор
      { dData = dd; }
// -------------------------------------------------------------
   public void displayLink()      // Вывод содержимого элемента
      { System.out.print(dData + " "); }
   }  // Конец класса Link
////////////////////////////////////////////////////////////////
```

```
class LinkList
   {
   private Link first;            // Ссылка на первый элемент в списке
// -------------------------------------------------------------
   public LinkList()              // Конструктор
      { first = null; }           // Список пока не содержит элементов
// -------------------------------------------------------------
   public Link getFirst()         // Получение первого элемента
      { return first; }
// -------------------------------------------------------------
   public void setFirst(Link f)   // Присваивание нового значения first
      { first = f; }
// -------------------------------------------------------------
   public boolean isEmpty()       // true, если список пуст
      { return first==null; }
// -------------------------------------------------------------
   public ListIterator getIterator()  // Получение итератора
      {
      return new ListIterator(this);  // Инициализация списком this
      }
// -------------------------------------------------------------
   public void displayList()
      {
      Link current = first;       // От начала списка
      while(current != null)      // Перемещение до конца списка
         {
         current.displayLink();   // Вывод текущего элемента
         current = current.next;  // Переход к следующему элементу
         }
      System.out.println("");
      }
// -------------------------------------------------------------
   }  // Конец класса LinkList
////////////////////////////////////////////////////////////////
class ListIterator
   {
   private Link current;          // Текущий элемент списка
   private Link previous;         // Предыдущий элемент списка
   private LinkList ourList;      // Связанный список
//--------------------------------------------------------------
   public ListIterator(LinkList list) // Конструктор
      {
      ourList = list;
      reset();
      }
//--------------------------------------------------------------
   public void reset()            // Возврат к 'first'
      {
      current = ourList.getFirst();
```

продолжение ⇨

Листинг 5.9 (*продолжение*)

```
      previous = null;
      }
//--------------------------------------------------------------
   public boolean atEnd()         // true, если текущим является
      { return (current.next==null); }    // последний элемент
//--------------------------------------------------------------
   public void nextLink()         // Переход к следующему элементу
      {
      previous = current;
      current = current.next;
      }
//--------------------------------------------------------------
   public Link getCurrent()       // Получение текущего элемента
      { return current; }
//--------------------------------------------------------------
   public void insertAfter(long dd)     // Вставка после
      {                                 // текущего элемента
      Link newLink = new Link(dd);

      if( ourList.isEmpty() )     // Пустой список
         {
         ourList.setFirst(newLink);
         current = newLink;
         }
      else                        // Список не пуст
         {
         newLink.next = current.next;
         current.next = newLink;
         nextLink();              // Переход к новому элементу
         }
      }
//--------------------------------------------------------------
   public void insertBefore(long dd)    // Вставка перед
      {                                 // текущим элементом
      Link newLink = new Link(dd);

      if(previous == null)              // В начале списка
         {                              // (или пустой список)
         newLink.next = ourList.getFirst();
         ourList.setFirst(newLink);
         reset();
         }
      else                              // Не в начале списка
         {
         newLink.next = previous.next;
         previous.next = newLink;
         current = newLink;
         }
      }
```

```
//--------------------------------------------------------------
   public long deleteCurrent()    // Удаление текущего элемента
      {
      long value = current.dData;
      if(previous == null)        // Если в начале списка
         {
         ourList.setFirst(current.next);
         reset();
         }
      else                        // Не в начале списка
         {
         previous.next = current.next;
         if( atEnd() )
            reset();
         else
            current = current.next;
         }
      return value;
      }
//--------------------------------------------------------------
   }  // Конец класса ListIterator
////////////////////////////////////////////////////////////////
class InterIterApp
   {
   public static void main(String[] args) throws IOException
      {
      LinkList theList = new LinkList();           // Создание списка
      ListIterator iter1 = theList.getIterator();  // Создание итератора
      long value;

      iter1.insertAfter(20);             // Вставка элементов
      iter1.insertAfter(40);
      iter1.insertAfter(80);
      iter1.insertBefore(60);

      while(true)
         {
         System.out.print("Enter first letter of show, reset, ");
         System.out.print("next, get, before, after, delete: ");
         System.out.flush();
         int choice = getChar();         // Ввод команды
         switch(choice)
            {
            case 's':                    // Вывод списка
               if( !theList.isEmpty() )
                  theList.displayList();
               else
                  System.out.println("List is empty");
               break;
```

продолжение ⇨

Листинг 5.9 (*продолжение*)

```
            case 'r':                    // Возврат к первому элементу
               iter1.reset();
               break;
            case 'n':                    // Переход к следующему элементу
               if( !theList.isEmpty() && !iter1.atEnd() )
                  iter1.nextLink();
               else
                  System.out.println("Can't go to next link");
               break;
            case 'g':                    // Получение текущего элемента
               if( !theList.isEmpty() )
                  {
                  value = iter1.getCurrent().dData;
                  System.out.println("Returned " + value);
                  }
               else
                  System.out.println("List is empty");
               break;
            case 'b':                    // Вставка перед текущим элементом
               System.out.print("Enter value to insert: ");
               System.out.flush();
               value = getInt();
               iter1.insertBefore(value);
               break;
            case 'a':                    // Вставка после текущего элемента
               System.out.print("Enter value to insert: ");
               System.out.flush();
               value = getInt();
               iter1.insertAfter(value);
               break;
            case 'd':                    // Удаление текущего элемента
               if( !theList.isEmpty() )
                  {
                  value = iter1.deleteCurrent();
                  System.out.println("Deleted " + value);
                  }
               else
                  System.out.println("Can't delete");
                  break;
            default:
               System.out.println("Invalid entry");
            }
         }
      }
//-------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
```

```
      String s = br.readLine();
      return s;
      }
//-------------------------------------------------------------
   public static char getChar() throws IOException
      {
      String s = getString();
      return s.charAt(0);
   }
//-------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
//-------------------------------------------------------------
   }  // Конец класса InterIterApp
////////////////////////////////////////////////////////////////
```

Метод main() вставляет четыре элемента в список, используя итератор и его метод insertAfter(), а затем ожидает ввода команд пользователем. В следующем примере пользователь выводит содержимое списка, возвращает итератор в начало, переходит на два элемента вперед, получает ключ текущего элемента (который равен 60), вставляет перед ним элемент 100, затем вставляет после 100 элемент с ключом 7 и снова выводит содержимое списка:

```
Enter first letter of
   show, reset, next, get, before, after, delete: s
20 40 60 80
Enter first letter of
   show, reset, next, get, before, after, delete: r
Enter first letter of
   show, reset, next, get, before, after, delete: n
Enter first letter of
   show, reset, next, get, before, after, delete: n
Enter first letter of
   show, reset, next, get, before, after, delete: g
Returned 60
Enter first letter of
   show, reset, next, get, before, after, delete: b
Enter value to insert: 100
Enter first letter of
   show, reset, next, get, before, after, delete: a
Enter value to insert: 7
Enter first letter of
   show, reset, next, get, before, after, delete: s
20 40 100 7 60 80
```

Поэкспериментируйте с программой interIterator.java — она даст вам представление о том, как итератор перемещается по элементам списка и как он вставляет/удаляет элементы в любой позиции списка.

На что указывает итератор?

При проектировании класса итератора необходимо принять одно важное решение: на какой элемент должен указывать итератор при выполнении различных операций?

Допустим, вы удалили элемент методом `deleteCurrent()`. Должен ли итератор после операции указывать на следующий элемент, на предыдущий элемент или вернуться к началу списка? Оставить итератор поблизости от удаленного элемента удобно — весьма вероятно, что пользователь класса будет выполнять следующие операции с этой позиции. Однако вернуть его к предыдущему элементу будет затруднительно, потому что не существует удобного способа занести ссылку на предыдущий элемент в поле `previous` списка (для этого нужен двусвязный список). Проблема решается перемещением итератора к элементу, следующему за удаленным. Если удаленный элемент находился в конце списка, итератор возвращается в начало списка.

Методы `insertBefore()` и `insertAfter()` возвращают значение `current`, указывающее на только что вставленный элемент.

Метод atEnd()

С методом `atEnd()` связана еще одна проблема. Метод может возвращать `true`, когда итератор указывает на последний действительный элемент списка или же тогда, когда итератор сместился «за» конец списка (то есть не указывает ни на какой действительный элемент).

В первом случае усложняется условие цикла, используемого для перебора элементов, потому что вам придется выполнить операцию с элементом перед проверкой того, является ли он последним (и завершить цикл, если является).

С другой стороны, во втором случае вы узнаете, что итератор находится в конце цикла, когда уже поздно выполнять какие-либо операции с последним элементом (например, вы не сможете найти последний элемент и удалить его). Когда `atEnd()` вернет `true`, итератор уже не указывает на последний элемент (и вообще на какой-либо действительный элемент), а в односвязном списке «отступить на шаг назад» уже не удастся.

Мы выбрали первый вариант. В нем итератор всегда указывает на действительную ссылку, хотя, как мы вскоре увидим, написание цикла с перебором списка потребует осторожности.

Итеративные операции

Как упоминалось ранее, итератор позволяет перебирать элементы списка и выполнять операции с элементами, отобранными по некоторому критерию. В следующем фрагменте для вывода содержимого списка используется итератор (вместо метода `displayList()` класса списка):

```
iter1.reset();                        // От начала списка
long value = iter1.getCurrent().dData;     // Вывод элемента
```

```
System.out.println(value + " ");
while( !iter1.atEnd() )                // Перемещение до конца списка
   {
   iter1.nextLink();                   // Переход к следующему элементу
   long value = iter1.getCurrent().dData;  // Вывод данных элемента
   System.out.println(value + " ");
   }
```

Хотя в приведенном фрагменте мы этого не делаем, перед вызовом `getCurrent()` следует проверить методом `isEmpty()`, что список не пуст.

Следующий фрагмент удаляет из списка все элементы с ключами, кратными 3. Приводится только измененный метод `main()`; все прочие методы остались такими же, как в программе `interIterator.java` (см. листинг 5.9).

```
class InterIterApp
   {
   public static void main(String[] args) throws IOException
      {
      LinkList theList = new LinkList();           // Создание списка
      ListIterator iter1 = theList.getIterator();  // Создание итератора

      iter1.insertAfter(21);             // Вставка элементов
      iter1.insertAfter(40);
      iter1.insertAfter(30);
      iter1.insertAfter(7);
      iter1.insertAfter(45);

      theList.displayList();             // Вывод содержимого списка

      iter1.reset();                     // От начала списка
      Link aLink = iter1.getCurrent();   // Получение текущего элемента
      if(aLink.dData % 3 == 0)           // Если ключ кратен 3,
         iter1.deleteCurrent();          // элемент удаляется.
      while( !iter1.atEnd() )            // Перемещение до конца списка
         {
         iter1.nextLink();               // Переход к следующему элементу

         aLink = iter1.getCurrent();     // Получение текущего элемента
         if(aLink.dData % 3 == 0)        // Если ключ кратен 3,
            iter1.deleteCurrent();       // элемент удаляется.
         }
      theList.displayList();             // Вывод содержимого списка
      }
   }  // Конец класса InterIterApp
```

Программа вставляет в список пять элементов и выводит его содержимое. Затем она перебирает элементы списка, удаляя элементы с ключами, кратными 3, и выводит список заново. Результат ее выполнения:

```
21 40 30 7 45
40 7
```

И снова, хотя здесь такая проверка отсутствует, перед вызовом `deleteCurrent()` следует проверять наличие элементов в списке.

Другие методы

Существуют и другие полезные методы, которые можно было бы определить для класса `ListIterator` — например, метод `find()`, возвращающий элемент с заданным ключом (аналог метода `find()` класса списка), или метод `replace()`, заменяющий элементы с заданным ключом другими элементами.

Так как список является односвязным, перебор по нему возможен только в прямом направлении. Использование двусвязного списка позволило бы переходить в обоих направлениях и выполнять такие операции, как удаление в конце списка. В некоторых ситуациях такая возможность может оказаться полезной.

Итоги

- Связанный список состоит из одного объекта `linkedList` и некоторого числа объектов `Link`.
- Объект `linkedList` содержит ссылку на первый элемент списка (которой часто присваивается имя `first`).
- Каждый объект `Link` содержит данные и ссылку на следующий элемент списка (которой часто присваивается имя `next`).
- Если поле `next` равно `null`, это является признаком конца списка.
- Чтобы вставить элемент в начале связанного списка, необходимо сохранить в поле `next` нового элемента ссылку, хранившуюся ранее в поле `first` списка, а в поле `first` — ссылку на новый элемент.
- Чтобы удалить элемент в начале списка, необходимо присвоить `first` ссылку на элемент `first.next`.
- Чтобы перемещаться по связанному списку, необходимо начать с элемента `first`, а затем последовательно переходить по ссылке `next` к следующему элементу списка.
- Поиск элемента с заданным ключом осуществляется перемещением по списку. Когда элемент будет найден, пользователь может вывести его содержимое, удалить или выполнить с ним другую операцию.
- После того как элемент с заданным значением ключа будет найден, можно вставить новый элемент до или после него.
- Двусторонний список хранит ссылку на последний элемент списка (которая часто называется `last`) наряду со ссылкой на первый элемент.
- В двустороннем списке возможна вставка элементов в конце списка.
- Абстрактным типом данных (ADT) называется класс, рассматриваемый независимо от его реализации.
- Стеки и очереди являются абстрактными типами данных. Они могут быть реализованы на базе как массивов, так и связанных списков.
- В сортированном связанном списке элементы упорядочиваются по возрастанию (а иногда по убыванию) значения ключа.

- ✦ Вставка в сортированном списке выполняется за время $O(N)$, потому что алгоритм должен найти правильную позицию для выполнения операции. Удаление наименьшего элемента выполняется за время $O(1)$.
- ✦ В двусвязном списка каждый элемент содержит ссылку как на следующий, так и на предыдущий элемент.
- ✦ Двусвязный список поддерживает перемещение в обратном направлении и удаление в конце списка.
- ✦ Итератор представляет собой ссылку, инкапсулированную в объекте класса, которая указывает на элемент ассоциированного списка.
- ✦ Методы итераторов обеспечивают перемещение итератора по списку и обращения к элементу, на который в данный момент указывает итератор.
- ✦ Итератор может использоваться для перемещения по списку с выполнением какой-либо операции с некоторыми (или всеми) элементами.

Вопросы

Следующие вопросы помогут читателю проверить качество усвоения материала. Ответы на них приведены в приложении В.

1. Какие из следующих утверждений *ложны*? Ссылка на объект класса:
   a) может использоваться для обращения к открытым методам объекта;
   b) обладает размером, зависящим от класса;
   c) относится к типу данных класса;
   d) не содержит сам объект класса.
2. Для обращения к элементам связанного списка обычно используется ссылка на ___________ элемент.
3. Создаваемая ссылка на элемент связанного списка:
   a) должна указывать на первый элемент списка;
   b) должна указывать на элемент, соответствующий `current`;
   c) должна указывать на элемент, соответствующий `next`;
   d) может указывать на произвольный элемент.
4. Сколько ссылок необходимо изменить для вставки элемента в середину односвязного списка?
5. Сколько ссылок необходимо изменить для вставки элемента в конец односвязного списка?
6. В коде метода `insertFirst()` программы `linkList.java` (см. листинг 5.1) команда `newLink.next=first;` означает, что:
   a) следующий создаваемый элемент будет содержать ссылку на `first`;
   b) в поле `first` заносится ссылка на новый элемент;

c) поле `next` нового элемента будет содержать ссылку на элемент, который раньше был первым в списке.

d) `newLink.next` будет содержать ссылку на новый первый элемент списка.

7. Предположим, поле `current` указывает на предпоследний элемент односвязного списка. Как выглядит команда для удаления последнего элемента списка?
8. Что происходит с элементом списка, когда все ссылки на него начинают указывать на другой объект?
9. Двусторонний список:

   a) поддерживает доступ с любого из двух концов;

   b) является другим названием для двусвязного списка;

   c) содержит указатели, обеспечивающие как прямое, так и обратное перемещение между элементами;

   d) связывает последний элемент с первым.
10. В методах вставки и удаления необходимо проверять особый случай: __________ список.
11. Если копирование занимает больше времени, чем сравнение, где удаление элемента с заданным ключом будет выполняться быстрее: в связанном списке или в несортированном массиве?
12. Сколько перемещений по односвязному списку потребуется для удаления элемента с наибольшим ключом?
13. Какая из разновидностей списков, рассмотренных в этой главе, лучше всего подойдет для реализации очереди?
14. Какое из следующих утверждений *ложно*? Итераторы удобно применить для выполнения:

    a) сортировки методом вставки в связанном списке;

    b) вставки нового элемента в начало списка;

    c) перестановки двух элементов в произвольных позициях;

    d) удаления всех элементов с заданным ключом.
15. Какая структура данных, на ваш взгляд, лучше подойдет для реализации стека: односвязный список или массив?

Упражнения

Упражнения помогут вам глубже усвоить материал этой главы. Программирования они не требуют.

1. Используйте приложение LinkList Workshop для выполнения операций вставки, поиска и удаления в сортированных и несортированных списках. Обладают ли сортированные списки какими-либо преимуществами для операций, продемонстрированных в этом приложении?

2. Измените метод `main()` в программе `linkList.java` (см. листинг 5.1) так, чтобы он постоянно вставлял элементы в список вплоть до исчерпания всей свободной памяти. После каждой тысячи элементов он должен выводить количество элементов, вставленных до настоящего момента. Это позволит вам примерно оценить максимальную емкость списка на вашем конкретном компьютере. (Конечно, число зависит от других программ, находящихся в памяти, и множества других факторов.) Не пытайтесь проводить этот эксперимент, если он может нарушить работу сети вашей организации.

Программные проекты

В этом разделе читателю предлагаются программные проекты, которые укрепляют понимание материала и демонстрируют практическое применение концепций этой главы.

5.1. Реализуйте приоритетную очередь на базе сортированного связанного списка. Операция извлечения из приоритетной очереди должна извлекать элемент с наименьшим ключом.

5.2. Реализуйте дек на базе двусвязного списка (см. программный проект 4.2 из предыдущей главы). Дек должен поддерживать все стандартные операции.

5.3. Циклическим списком называется связанный список, в котором последний элемент содержит ссылку на первый элемент. Существует много способов реализации циклических списков. Иногда объект списка содержит указатель на «начало» списка. Однако в этом случае список уже не похож на замкнутый круг, а больше напоминает обычный список, у которого конец связан с началом. Создайте класс для односвязного списка, не имеющий ни начала, ни конца. Доступ к списку осуществляется по единственной ссылке `current`, которая может указывать на любой элемент списка. Ссылка перемещается по списку так, как требуется пользователю (ситуация, для которой идеально подходит циклический список, представлена в проекте 5.5). Список должен поддерживать операции вставки, поиска и удаления. Вероятно, удобнее выполнять эти операции с элементом, следующим за тем, на который указывает `current`. (Так как список является односвязным, вы не сможете вернуться к предыдущему элементу без полного прохождения всей цепочки.) Также следует предусмотреть возможность вывода содержимого списка (хотя его придется разбить в некоторой точке для вывода на экран). Метод `step()`, который перемещает `current` к следующему элементу, тоже может пригодиться.

5.4. Реализуйте класс стека на базе циклического списка из проекта 5.3. Этот проект не слишком сложен. (Впрочем, реализовать очередь будет уже сложнее, если только не перейти к двусвязному списку.)

5.5. Задача Иосифа Флавия — известная математическая задача с историческим подтекстом. Существует много легенд о том, как она возникла. В одной из них говорится, что Иосиф был в составе гарнизона осажденной крепости, которую должны были захватить римляне. Защитники крепости предпочли смерть рабству. Они выстроились в круг, и начинали считать по кругу, выбрав в качестве начала

отсчета конкретного человека. Каждый n-й по счету покидал круг и совершал самоубийство. Иосиф решил, что ему умирать еще рано, и выбрал такие правила, чтобы остаться последним из выживших. Если в круге было (например) 20 человек и он был седьмым от начала отсчета, какое число следовало выбрать для отсчета? Задача сильно усложняется тем, что круг уменьшается по мере выбывания участников.

Создайте приложение, моделирующее задачу с использованием циклического связанного списка (как в проекте 5.3). Во входных параметрах передаются количество людей в круге, число для отсчета и номер человека, с которого начинается отсчет (чаще всего 1). Программа выводит список выбывающих. Когда один из участников выходит из круга, отсчет начинается заново слева от него (предполагается, что отсчет ведется по часовой стрелке). Например, если в круге стоят семеро участников с номерами от 1 до 7, а отсчет ведется через 3 позиции (начиная с позиции 1), то участники выбывают в порядке 4, 1, 6, 5, 7, 3. Номер 2 остается последним.

5.6. Попробуйте реализовать несколько иную структуру данных: двумерный связанный список, который мы будем называть матрицей. Она представляет собой списковый аналог двумерных массивов и может использоваться в различных приложениях, например в электронных таблицах. Если электронная таблица построена на базе массива, то вставка нового элемента в начале потребует перемещения каждой из $N \times M$ нижележащих ячеек, что, вероятно, займет много времени. Если электронная таблица реализована на базе матрицы, то при вставке достаточно будет изменить N указателей.

Для простоты будет использоваться односвязный список (хотя двусвязный список в данном случае был бы более уместным). На каждый элемент (кроме находящихся в верхней строке и левом столбце) указывают ссылки элементов, находящихся непосредственно над ним и слева от него. Скажем, можно начать с левого верхнего угла и перейти к элементу в третьей строке и пятом столбце — для этого достаточно переместиться на две строки вниз и четыре столбца вправо. Матрица создается с изначально заданными размерами (допустим, 7 × 10). Ваша реализация должна поддерживать вставку значений в заданных позициях и вывод текущего содержимого матрицы.

Глава 6
Рекурсия

Рекурсией называется методология программирования, при которой метод (функция) вызывает сам себя. На первый взгляд такой вызов выглядит довольно странно и даже кажется катастрофической ошибкой. Однако на практике рекурсия оказывается одной из самых интересных и неожиданно эффективных методологий программирования. При первом знакомстве она кажется чем-то невероятным — вроде подвига барона Мюнхгаузена, вытащившего себя за волосы из болота. Тем не менее рекурсия не только работает, но и предоставляет уникальную концептуальную основу для решения многих задач.

В этой главе рассматриваются многочисленные примеры ситуаций, в которых уместно применение рекурсии. Мы займемся вычислением треугольных чисел и факториалов, построением анаграмм, выполнением рекурсивного двоичного поиска, решением головоломки «Ханойская башня» и изучим один из методов сортировки, называемый «сортировкой слиянием». Для «Ханойской башни» и сортировки слиянием имеются демонстрационные приложения.

Также мы рассмотрим достоинства и недостатки рекурсии и возможность преобразования рекурсивных решений в стековые.

Треугольные числа

Говорят, пифагорейцы — группа древнегреческих математиков, работавших под началом Пифагора (автора знаменитой теоремы) — ощущали мистическую связь с числовыми рядами вида 1, 3, 6, 10, 15, 21, ... Сможете ли вы угадать следующее число в этом ряду?

N-е число ряда получается прибавлением *n* к предыдущему числу. Таким образом, чтобы получить второе число, мы увеличиваем первое (1) на 2; 1 + 2 = 3. Третье число получается увеличением второго (3) на 3; 3 + 3 = 6 и т. д.

Такие ряды называются *треугольными числами*, потому что их можно наглядно представить как число объектов, которые могут быть расставлены в форме треугольника (рис. 6.1).

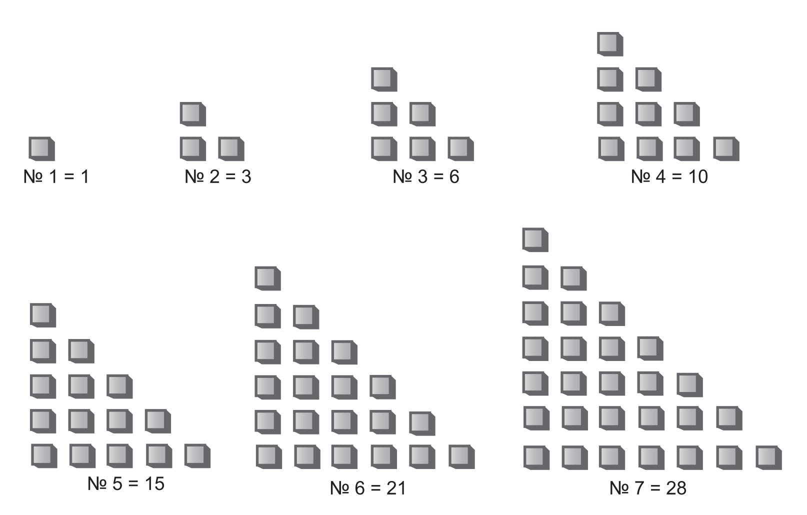


Рис. 6.1. Треугольные числа

Вычисление n-го треугольного числа в цикле

Допустим, вы хотите определить значение произвольного *n*-го числа в серии — допустим, 4-го (10). Как его вычислить? Из рис. 6.2 видно, что значение любого числа ряда вычисляется суммированием величин всех вертикальных столбцов.

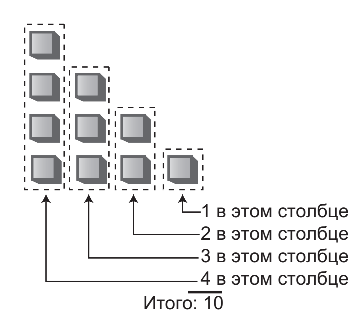


Рис. 6.2. Вычисление треугольных чисел

Для четвертого числа первый столбец состоит из четырех квадратиков, второй — из трех и т. д. Суммирование 4 + 3 + 2 + 1 дает 10.

В приведенном ниже методе `triangle()` суммирование по столбцам используется для вычисления треугольных чисел. Метод суммирует высоты всех столбцов от *n* до 1:

```
int triangle(int n)
   {
   int total = 0;

   while(n > 0)           // Пока n равно 1 и более
```

```
    {
    total = total + n;  // Переменная total увеличивается на n (высоту столбца)
    --n;                // Уменьшение высоты столбца
    }
  return total;
  }
```

Метод выполняет n итераций цикла. В первой итерации `total` увеличивается на n, во второй — на $n - 1$ и так далее до 1. Когда значение n становится равно 0, цикл прерывается.

Вычисление n-го треугольного числа с применением рекурсии

Решение с циклом получается простым и доступным, но существует и другой подход к решению этой задачи. Значение n-го числа ряда может рассматриваться как сумма только двух слагаемых (вместо суммы целого ряда):

1. Первый (самый высокий) столбец со значением n.
2. Сумма всех остальных столбцов.

Концепция представлена на рис. 6.3.

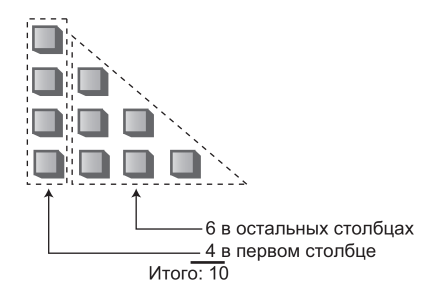


Рис. 6.3. Треугольное число как сумма высоты одного столбца и треугольника

Вычисление суммы остальных столбцов

Если бы в нашем распоряжении был метод вычисления суммы всех остальных столбцов, то метод `triangle()`, вычисляющий значение n-го треугольного числа, можно было бы записать в следующем виде:

```
int triangle(int n)
  {
  return( n + sumRemainingColumns(n) );  // (Предварительная версия)
  }
```

Но чего мы добились? Написать метод `sumRemainingColumns()` ничуть не проще, чем исходный метод `triangle()`.

Обратите внимание: на рис. 6.3 сумма остальных столбцов для числа n равна сумме всех столбцов для числа $n - 1$. Таким образом, если бы у нас был метод,

суммирующий все столбцы для числа *n*, мы могли бы вызвать его с аргументом *n* – 1 для вычисления суммы всех остальных столбцов для числа *n*:

```
int triangle(int n)
   {
   return( n + sumAllColumns(n-1) );  // (Предварительная версия)
   }
```

Но ведь если задуматься, метод `sumAllColumns()` делает точно то же самое, что и метод `triangle()`: он суммирует все столбцы для числа *n*, переданного в аргументе. Так почему бы не заменить его самим методом `triangle()`? Результат будет выглядеть так:

```
int triangle(int n)
   {
   return( n + triangle(n-1) );  // (Предварительная версия)
   }
```

Возможно, вас удивит, что метод вызывает сам себя, но, собственно, почему бы и нет? Вызов метода представляет собой (в частности) передачу управления в начало метода. Передача управления может происходить как из самого метода, так и снаружи.

Эстафета

Рекурсивное решение напоминает эстафетную передачу. Кто-то приказывает вам вычислить 9-е треугольное число. Вы знаете, что оно равно сумме 9-го и 8-го треугольного числа, поэтому вы вызываете Гарри и приказываете ему вычислить 8-е треугольное число. Получив от него ответ, вы прибавляете к нему 9 и получаете конечный результат.

Гарри знает, что 8-е треугольное число равно сумме 8-го и 7-го треугольного числа. Он вызывает Салли и приказывает ей вычислить 7-е треугольное число. Процесс продолжается, а каждый участник передает эстафету следующему.

Когда кончается передача? В определенный момент кто-то сможет вычислить ответ, не обращаясь за помощью. Если бы этого не произошло, то возникла бы бесконечная цепочка людей, обращающихся друг к другу с запросами — математическая «пирамида», которая никогда не кончается. Для метода `triangle()` это означало бы, что метод бесконечно вызывает сам себя, пока программа не завершится сбоем.

Прекращение эстафеты

Для предотвращения бесконечной передачи человек, которому приказано вычислить первое треугольное число ($n = 1$), должен *знать*, что ответ равен 1 — для получения этого ответа он ни к кому не обращается с запросом. Минимальное число достигнуто, поэтому передача эстафеты прекращается. Ситуация выражается включением в метод `triangle()` дополнительного условия:

```
int triangle(int n)
   {
   if(n==1)
      return 1;
```

```
   else
      return( n + triangle(n-1) );
   }
```

Условие, приводящее к возврату управления рекурсивным методом без следующего рекурсивного вызова, называется *базовым ограничением*. Каждый рекурсивный метод должен иметь базовое ограничение для предотвращения бесконечной рекурсии и последующего аварийного завершения программы.

Программа triangle.java

Запустив программу `triangle.java`, вы сможете убедиться в том, что рекурсия действительно работает. Введите номер треугольного числа *n* — программа выведет его значение. Код `triangle.java` приведен в листинге 6.1.

Листинг 6.1. Программа triangle.java

```
// triangle.java
// Вычисление треугольных чисел
// Запуск программы: C>java TriangleApp
import java.io.*;                 // Для ввода/вывода
////////////////////////////////////////////////////////////////
class TriangleApp
   {
   static int theNumber;

   public static void main(String[] args) throws IOException
      {
      System.out.print("Enter a number: ");
      theNumber = getInt();
      int theAnswer = triangle(theNumber);
      System.out.println("Triangle="+theAnswer);
   }
//-------------------------------------------------------------
   public static int triangle(int n)
      {
      if(n==1)
         return 1;
      else
         return( n + triangle(n-1) );
      }
//-------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
```

продолжение ⇨

Листинг 6.1 (*продолжение*)

```
//-------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
//-------------------------------------------------------------
   }  // Конец класса TriangleApp
///////////////////////////////////////////////////////////////
```

Метод main() запрашивает у пользователя номер треугольного числа, вызывает triangle() и выводит возвращаемое значение. Вся реальная работа выполняется методом triangle(), который многократно вызывает сам себя.

Примерный результат выполнения:

```
Enter a number: 1000
Triangle = 500500
```

А если вы вдруг усомнитесь в результатах, возвращаемых triangle(), проверьте их по формуле:

```
n-е треугольное число = (n2+n)/2
```

Что реально происходит?

Давайте слегка изменим метод triangle(), чтобы он выдавал информацию о том, что происходит во время его выполнения. Дополнительные команды вывода помогут следить за аргументами и возвращаемыми значениями:

```
public static int triangle(int n)
   {
   System.out.println("Entering: n=" + n);
   if(n==1)
      {
      System.out.println("Returning 1");
      return 1;
      }
   else
      {
      int temp = n + triangle(n-1);
      System.out.println("Returning " + temp);
      return temp;
      }
   }
```

Вот как выглядит результат выполнения программы для новой версии метода triangle(), когда пользователь вводит число 5:

```
Enter a number: 5
Entering: n=5
Entering: n=4
```

```
Entering: n=3
Entering: n=2
Entering: n=1
Returning 1
Returning 3
Returning 6
Returning 10
Returning 15

Triangle = 15
```

Каждый раз, когда метод `triangle()` вызывает сам себя, его аргумент, изначально равный 5, уменьшается на 1. Рекурсивные вызовы продолжаются до тех пор, пока аргумент не уменьшится до 1, после чего метод возвращает значение. Это приводит к целой серии каскадных возвращений. Метод, словно феникс из пепла, восстает из отработанных версий самого себя. При каждом возвращении значение *n*, с которым метод был вызван, прибавляется к возвращаемому значению вызванного метода.

Возвращаемые значения воспроизводят ряд треугольных чисел до тех пор, пока ответ не вернется в метод `main()`. На рис. 6.4 показано, как каждый рекурсивный вызов `triangle()` осуществляется «изнутри» предыдущего вызова.

Непосредственно перед тем, как внутренняя версия возвращает 1, в программе одновременно существуют пять вызовов `triangle()`. Внешний вызов получает аргумент 5, а внутренний — аргумент 1.

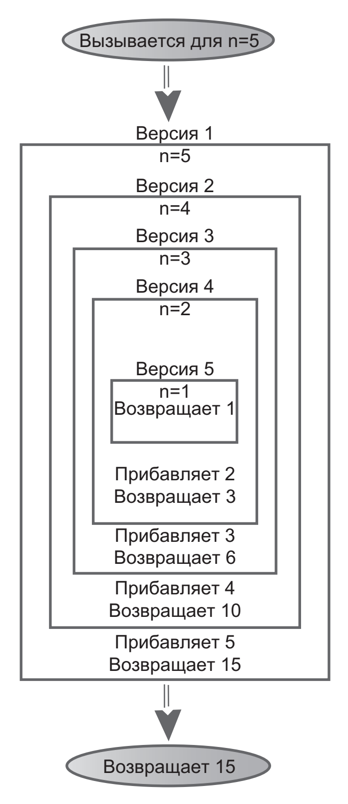


Рис. 6.4. Рекурсивный метод triangle()

Характеристики рекурсивных методов

Метод `triangle()`, несмотря на простоту, обладает всеми характеристиками рекурсивных методов:

- Он вызывает сам себя.
- Рекурсивный вызов предназначен для решения упрощенной задачи.
- Существует версия задачи, достаточно простая для того, чтобы метод мог решить ее и вернуть управление без рекурсии.

При каждом последующем вызове рекурсивного метода значение аргумента уменьшается (или диапазон, определяемый несколькими аргументами, сужается) — в этом проявляется постепенное «упрощение» задачи. Когда аргумент или диапазон достигает определенного минимального размера, срабатывает базовое ограничение, и метод возвращает управление без рекурсии.

Насколько эффективна рекурсия?

Вызов метода сопряжен с определенными непроизводительными затратами ресурсов. Управление должно передаваться из текущей точки вызова в начало метода. Кроме того, аргументы метода и адрес возврата заносятся во внутренний стек, чтобы метод мог обратиться к значениям аргументов и знать, по какому адресу следует вернуть управление.

В ситуации с методом `triangle()` вполне вероятно, что из-за этих затрат решение с циклом будет работать быстрее рекурсивного решения. Отставание может быть незначительным, но при большом количестве вызовов желательно избавиться от рекурсии. Эта тема более подробно рассматривается в конце главы.

Другой фактор снижения эффективности — затраты памяти на хранение всех промежуточных аргументов и возвращаемых значений во внутреннем стеке. Большой объем данных может создать проблемы и привести к переполнению стека.

Рекурсия обычно применяется ради концептуального упрощения задачи, а не оттого, что она изначально более эффективна.

Математическая индукция

Рекурсия является программным аналогом математической индукции — способа определения чего-либо в контексте определяемой сущности. (Этим термином также обозначается способ доказательства теорем.) Индукция позволяет определять треугольные числа на математическом уровне:

```
tri(n) = 1                    if n = 1
tri(n) = n + tri(n-1)         if n > 1
```

Может показаться, что определение чего-либо в контексте определяемой сущности создает порочный круг, но в действительности оно абсолютно законно (при наличии базового ограничения).

Факториал

Факториалы сходны с треугольными числами, если не считать того, что вместо сложения используется операция умножения. Треугольное число с номером n вычисляется суммированием треугольного числа $n - 1$ с n, а факториал n вычисляется умножением факториала $n - 1$ на n. Иначе говоря, пятое треугольное число вычисляется по формуле $5 + 4 + 3 + 2 + 1$, а факториал 5 вычисляется по формуле $5 \times 4 \times 3 \times 2 \times 1 = 120$. В таблице 6.1 перечислены факториалы первых 10 чисел.

Таблица 6.1. Факториалы

Число	Формула	Факториал
0	По определению	1
1	1 * 1	1

Число	Формула	Факториал
2	2 * 1	2
3	3 * 2	6
4	4 * 6	24
5	5 * 24	120
6	6 * 120	720
7	7 * 720	5 040
8	8 * 5 040	40 320
9	9 * 40 320	362 880

Факториал 0 по определению равен 1. Как видно из таблицы, факториалы растут очень быстро.

Для вычисления факториала можно использовать рекурсивный метод, аналогичный методу triangle(). Он выглядит примерно так:

```
int factorial(int n)
   {
   if(n==0)
      return 1;
   else
      return (n * factorial(n-1) );
   }
```

Между методами factorial() и triangle() есть только два отличия. Во-первых, factorial() использует * вместо + в выражении

```
n * factorial(n-1)
```

Во-вторых, базовое ограничение в factorial() выполняется, когда значение n равно 0, а не 1. Пример взаимодействия с этим методом в программе, сходной с triangle.java:

```
Enter a number: 6
Factorial =720
```

На рис. 6.5 показано, как различные варианты вызовов factorial() передают управление друг другу при исходном вызове с $n = 4$.

Вычисление факториала является классическим примером рекурсии, хотя факториалы не имеют такого наглядного представления, как треугольные числа.

Рекурсивные методы также могут применяться при решении других вычислительных задач, например вычисления наибольшего общего кратного двух чисел (используемого для сокращения дробей), возведения числа в степень и т. д. Впрочем, как и в пре-

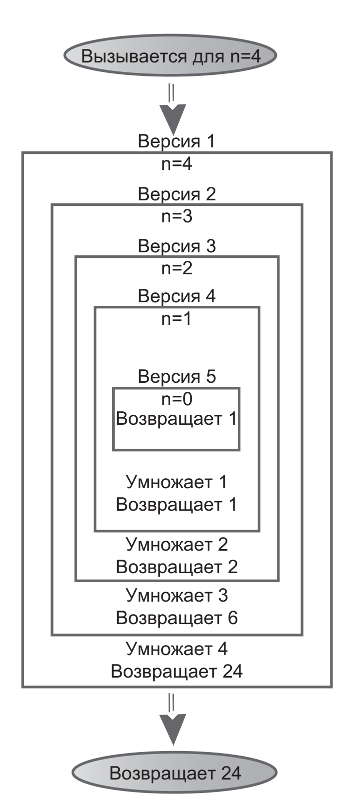


Рис. 6.5. Рекурсивный метод factorial()

дыдущих случаях, эти решения хорошо подходят для демонстрации рекурсии, но на практике решения на базе циклов обычно оказываются более эффективными.

Анаграммы

Рассмотрим другую область, в которой рекурсия предоставляет элегантное решение задачи. *Перестановкой* называется расположение объектов в определенном порядке. Предположим, требуется составить полный список анаграмм заданного слова, то есть всех возможных перестановок букв (независимо от того, являются они допустимыми словами или нет). Например, для слова *cat* программа должна вывести следующий список анаграмм:

```
cat
cta
atc
act
tca
tac
```

Попробуйте самостоятельно составить анаграммы некоторых слов. Вскоре становится ясно, что количество вариантов равно факториалу количества букв. Для слова из трех букв существует 6 анаграмм, для слова из четырех букв — 24 анаграммы, для слова из пяти букв — 120 анаграмм и т. д. (Предполагается, что буквы в слове не повторяются; при повторении букв количество возможных слов будет меньше.)

Как написать программу для построения списка анаграмм слова? Один из возможных способов для слова из n букв:

1. Построить анаграммы для правых $n-1$ букв.
2. Выполнить циклический сдвиг всех n букв.
3. Повторить эти действия n раз.

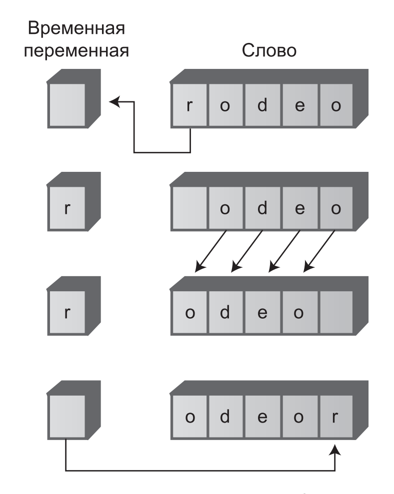


Рис. 6.6. Циклический сдвиг букв слова

При циклическом сдвиге все буквы сдвигаются на одну позицию влево — кроме крайней левой буквы, которая перемещается в конец слова (рис. 6.6).

Если выполнить циклический сдвиг слова *n* раз, каждая из букв слова побывает первой. Пока выбранная буква занимает первую позицию, для всех остальных букв строится список анаграмм (то есть буквы переставляются во всех возможных сочетаниях). Слово *cat* состоит всего из трех букв, поэтому циклический сдвиг двух последних букв просто меняет их местами. Последовательность действий показана в табл. 6.2.

Обратите внимание на необходимость возврата к исходному расположению двух букв перед выполнением циклического сдвига всех трех букв. По этой причине в серии генерируются последовательности вида *cat*, *cta*, *cat*. Лишние анаграммы не отображаются.

Как строится список анаграмм правых $n - 1$ букв? Посредством рекурсивного вызова. Рекурсивный метод `doAnagram()` получает единственный параметр с размером слова, для которого строятся анаграммы. Предполагается, что это слово содержит правые *n* букв исходного слова. Каждый раз, когда метод `doAnagram()` рекурсивно вызывает себя, он делает это для уменьшенного на единицу количества букв (рис. 6.7).

Таблица 6.2. Анаграммы слова cat

Слово	Выводить слово?	Первая буква	Остальные буквы	Действие
cat	Да	c	at	Циклический сдвиг at
cta	Да	c	ta	Циклический сдвиг ta
cat	Нет	c	at	Циклический сдвиг cat
atc	Да	a	tc	Циклический сдвиг tc
act	Да	a	ct	Циклический сдвиг ct
atc	Нет	a	tc	Циклический сдвиг atc
tca	Да	t	ca	Циклический сдвиг ca
tac	Да	t	ac	Циклический сдвиг ac
tca	Нет	t	ca	Циклический сдвиг tca
cat	Нет	c	at	Завершение

Базовое ограничение встречается тогда, когда размер слова, для которого строятся анаграммы, составляет всего одну букву. Сгенерировать для него новые перестановки нельзя, поэтому метод немедленно возвращает управление. В противном случае он строит анаграммы всех букв, кроме первой буквы переданного слова, и выполняет циклический сдвиг всего слова. Эти два действия выполняются *n* раз, где *n* — длина слова. Код рекурсивного метода `doAnagram()`:

```
public static void doAnagram(int newSize)
   {
   if(newSize == 1)                  // Если слово слишком маленькое,
      return;                        // не продолжать.
   for(int j=0; j<newSize; j++)      // Для каждой позиции
```

```
    {
    doAnagram(newSize-1);         // Построить анаграммы остальных букв
    if(newSize==2)                // Если внутреннее состояние
       displayWord();             // Вывести слово
    rotate(newSize);              // Циклический сдвиг всего слова
    }
 }
```

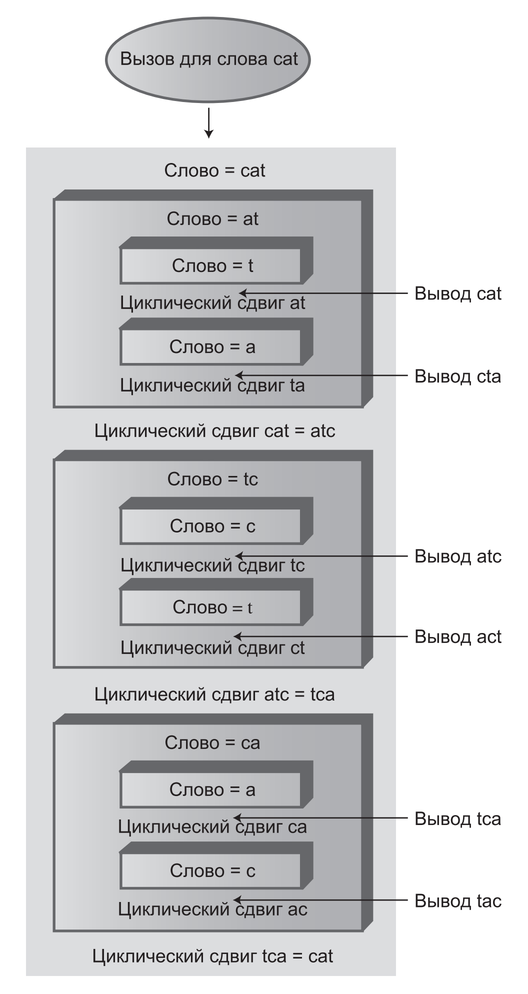


Рис. 6.7. Рекурсивный метод doAnagram()

Каждый раз, когда метод doAnagram() вызывает себя, размер слова уменьшается на одну букву, а начальная позиция сдвигается на одну ячейку вправо (рис. 6.8).

В листинге 6.2 приведен полный код программы anagram.java. Метод main() запрашивает слово у пользователя, сохраняет его в символьном массиве для удобства работы, после чего вызывает doAnagram().

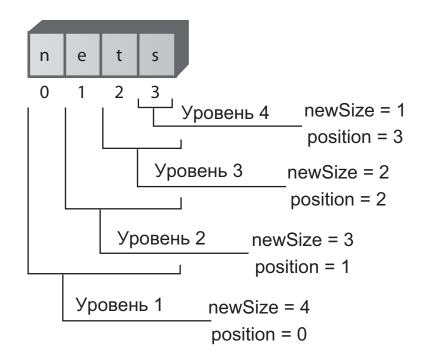


Рис. 6.8. Последовательное уменьшение слова

Листинг 6.2. Программа anagram.java

```
// anagram.java
// Построение списка анаграмм
// Запуск программы: C>java AnagramApp
import java.io.*;
////////////////////////////////////////////////////////////////
class AnagramApp
   {
   static int size;
   static int count;
   static char[] arrChar = new char[100];

   public static void main(String[] args) throws IOException
      {
      System.out.print("Enter a word: ");    // Получение слова
      String input = getString();
      size = input.length();                 // Определение размера
      count = 0;
      for(int j=0; j<size; j++)              // Сохранение в массиве
         arrChar[j] = input.charAt(j);
      doAnagram(size);                       // Построение анаграмм
   }
//-------------------------------------------------------------
   public static void doAnagram(int newSize)
      {
      if(newSize == 1)                 // Если слово слишком маленькое,
         return;                       // не продолжать.
      for(int j=0; j<newSize; j++)     // Для каждой позиции
         {
         doAnagram(newSize-1);         // Построить анаграммы остальных букв
         if(newSize==2)                // Если внутреннее состояние
            displayWord();             // Вывести слово
         rotate(newSize);              // Циклический сдвиг всего слова
         }
      }
```

продолжение ⇗

Листинг 6.2 (*продолжение*)

```
//-------------------------------------------------------------
   // rotate left all chars from position to end
   public static void rotate(int newSize)
      {
      int j;
      int position = size - newSize;
      char temp = arrChar[position];       // Сохранение первой буквы
      for(j=position+1; j<size; j++)       // Сдвиг остальных букв влево
         arrChar[j-1] = arrChar[j];
      arrChar[j-1] = temp;                 // Перемещение первой буквы
      }                                    // на правый край
//-------------------------------------------------------------
   public static void displayWord()
      {
      if(count < 99)
         System.out.print(" ");
      if(count < 9)
         System.out.print(" ");
      System.out.print(++count + " ");
      for(int j=0; j<size; j++)
         System.out.print( arrChar[j] );
      System.out.print("   ");
      System.out.flush();
      if(count%6 == 0)
         System.out.println("");
      }
//-------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   }  // Конец класса AnagramApp
//////////////////////////////////////////////////////////////
```

Метод rotate() осуществляет циклический сдвиг на одну позицию влево так, как было описано ранее. Метод displayWord() выводит все слово и добавляет счетчик, чтобы пользователю стало понятно, сколько слов было выведено. Пример работы программы:

```
Enter a word: cats
1 cats     2 cast     3 ctsa     4 ctas     5 csat     6 csta
7 atsc     8 atcs     9 asct    10 astc    11 acts    12 acst
13 tsca    14 tsac    15 tcas    16 tcsa    17 tasc    18 tacs
19 scat    20 scta    21 satc    22 sact    23 stca    24 stac
```

Программа может использоваться для построения анаграмм пяти- и даже шестибуквенных слов. Но поскольку факториал 6 равен 720, для длинных слов программа выдаст больше анаграмм, чем вам хотелось бы.

Рекурсивный двоичный поиск

Помните двоичный поиск, описанный в главе 2, «Массивы»? Алгоритм предназначался для поиска заданного элемента упорядоченного массива с минимальным числом сравнений. Массив делился пополам, алгоритм определял, в какой половине находится нужный элемент, после чего эта половина снова делилась пополам и т. д. Исходная версия метода `find()` выглядела так:

```
public int find(long searchKey)
   {
   int lowerBound = 0;
   int upperBound = nElems-1;
   int curIn;

   while(true)
      {
      curIn = (lowerBound + upperBound ) / 2;
      if(a[curIn]==searchKey)
         return curIn;                    // Элемент найден
      else if(lowerBound > upperBound)
         return nElems;                   // Элемент не найден
      else                                // Деление диапазона
         {
         if(a[curIn] < searchKey)
            lowerBound = curIn + 1;       // В верхней половине
         else
            upperBound = curIn - 1;       // В нижней половине
         }
      }
   }
```

Перечитайте раздел главы 2, посвященный двоичному поиску в упорядоченных массивах, — в нем рассказано, как работает этот метод. Если хотите увидеть двоичный поиск в действии, запустите приложение Ordered Workshop этой главы.

Метод, использующий цикл, достаточно легко преобразуется в рекурсивный метод. В циклической версии новый диапазон определяется посредством изменения `lowerBound` или `upperBound`, после чего цикл выполняется заново. При каждой итерации диапазон делится (приблизительно) надвое.

Замена цикла рекурсией

В рекурсивной версии вместо изменения lowerBound или upperBound метод find() вызывается заново для новых значений параметров lowerBound или upperBound. Цикл исключается из программы и заменяется рекурсивными вызовами. Вот как это выглядит:

```
private int recFind(long searchKey, int lowerBound,
                                    int upperBound)
   {
   int curIn;

   curIn = (lowerBound + upperBound ) / 2;
   if(a[curIn]==searchKey)
      return curIn;                                    // Элемент найден
   else if(lowerBound > upperBound)
      return nElems;                                   // Элемент не найден
   else                                                // Деление диапазона
      {
      if(a[curIn] < searchKey)                         // В верхней половине
         return recFind(searchKey, curIn+1, upperBound);
      else                                             // В нижней половине
         return recFind(searchKey, lowerBound, curIn-1);
      }
   }
```

Пользователь класса (представленный методом main()) при вызове find() может не знать, сколько элементов содержит массив; в любом случае его не следует обременять поиском начальных значений upperBound и lowerBound. По этой причине в программе определяется вспомогательный открытый метод find(), который вызывается методом main() с единственным аргументом — искомым значением ключа. Метод find() задает правильные исходные значения lowerBound и upperBound (0 и nElems-1), после чего вызывает приватный рекурсивный метод recFind(). Метод find() выглядит так:

```
public int find(long searchKey)
   {
   return recFind(searchKey, 0, nElems-1);
   }
```

Полный код программы binarySearch.java приведен в листинге 6.3.

Листинг 6.3. Программа binarySearch.java

```
// binarySearch.java
// Рекурсивный двоичный поиск
// Запуск программы: C>java BinarySearchApp
////////////////////////////////////////////////////////////////
class ordArray
   {
   private long[] a;                 // Ссылка на массив a
   private int nElems;               // number of data items
```

```
//-----------------------------------------------------------
   public ordArray(int max)          // Конструктор
      {
      a = new long[max];             // Создание массива
      nElems = 0;
      }
//-----------------------------------------------------------
   public int size()
      { return nElems; }
//-----------------------------------------------------------
   public int find(long searchKey)
      {
      return recFind(searchKey, 0, nElems-1);
      }
//-----------------------------------------------------------
   private int recFind(long searchKey, int lowerBound,
                                       int upperBound)
      {
      int curIn;

      curIn = (lowerBound + upperBound ) / 2;
      if(a[curIn]==searchKey)
         return curIn;              // Элемент найден
      else if(lowerBound > upperBound)
         return nElems;             // Элемент не найден
      else                          // Деление диапазона
         {
         if(a[curIn] < searchKey)   // В верхней половине
            return recFind(searchKey, curIn+1, upperBound);
         else                       // В нижней половине
            return recFind(searchKey, lowerBound, curIn-1);
         }
      }
//-----------------------------------------------------------
   public void insert(long value)    // Сохранение элемента в массиве
      {
      int j;
      for(j=0; j<nElems; j++)        // Определение позиции
         if(a[j] > value)            // (линейный поиск)
            break;
      for(int k=nElems; k>j; k--)    // Перемещение элементов
         a[k] = a[k-1];              // с большим значением ключа
      a[j] = value;                  // Вставка
      nElems++;                      // Увеличение размера
      }
//-----------------------------------------------------------
   public void display()             // Вывод содержимого массива
      {
      for(int j=0; j<nElems; j++)       // Для каждого элемента
```

продолжение ☞

Листинг 6.3 (*продолжение*)

```
         System.out.print(a[j] + " ");  // Вывод текущего элемента
      System.out.println("");
   }
//-------------------------------------------------------------
}  // Конец класса ordArray
////////////////////////////////////////////////////////////////
class BinarySearchApp
   {
   public static void main(String[] args)
      {
      int maxSize = 100;             // Размер массива
      ordArray arr;                  // Ссылка на массив
      arr = new ordArray(maxSize);   // Создание массива

      arr.insert(72);                // Вставка элементов
      arr.insert(90);
      arr.insert(45);
      arr.insert(126);
      arr.insert(54);
      arr.insert(99);
      arr.insert(144);
      arr.insert(27);
      arr.insert(135);
      arr.insert(81);
      arr.insert(18);
      arr.insert(108);
      arr.insert(9);
      arr.insert(117);
      arr.insert(63);
      arr.insert(36);

      arr.display();                 // Вывод содержимого массива

      int searchKey = 27;            // Поиск элемента
      if( arr.find(searchKey) != arr.size() )
         System.out.println("Found " + searchKey);
      else
         System.out.println("Can't find " + searchKey);
      }
   }  // Конец класса BinarySearchApp
////////////////////////////////////////////////////////////////
```

Метод `main()` вставляет в массив 16 элементов. Метод `insert()` расставляет их в порядке сортировки, после чего программа выводит отсортированный массив. Наконец, метод `find()` используется для поиска элемента с ключом 27. Пример вывода:

```
9 18 27 36 45 54 63 72 81 90 99 108 117 126 135 144
Found 27
```

В программе binarySearch.java массив содержит 16 элементов. На рис. 6.9 показано, как метод recFind() этой программы вызывает себя снова и снова для сужающегося диапазона. Когда «самая внутренняя» версия метода находит искомый элемент с ключом 27, она возвращает его индекс 2 (как видно из вывода упорядоченного массива). Затем это значение последовательно возвращается каждой версией recFind(); наконец, метод find() передает его пользователю класса.

Рекурсивный двоичный поиск обладает такой же сложностью, как и нерекурсивный: $O(\log N)$. Решение выглядит немного элегантнее, но может работать чуть медленнее.

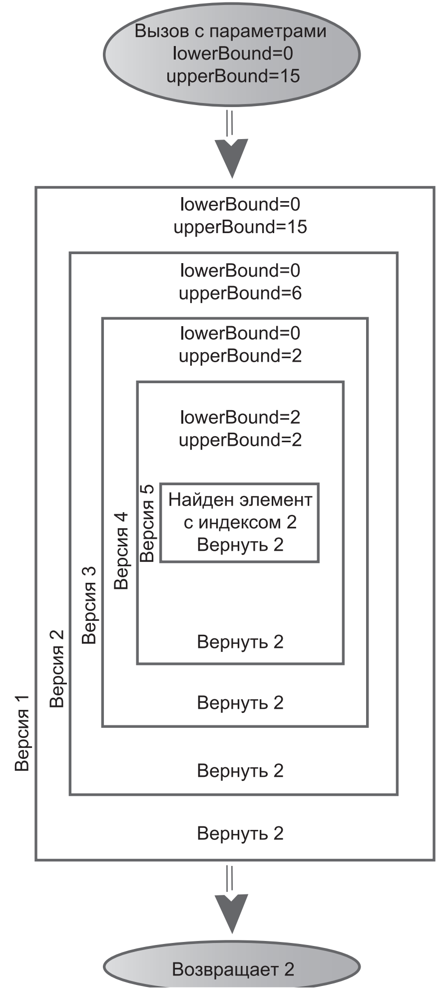


Рис. 6.9. Рекурсивный метод binarySearch()

Алгоритмы последовательного разделения

Рекурсивный двоичный поиск является примером *алгоритма последовательного разделения*. Бóльшая задача делится на две меньшие задачи, решаемые по отдельности. Меньшие задачи решаются одинаково: каждая из них делится на две еще меньшие задачи. Процесс продолжается до тех пор, пока не доберется до базового ограничения, которое тривиально решается без дальнейшего деления.

Метод последовательного разделения часто применяется в сочетании с рекурсией, хотя, как было показано при описании двоичного поиска в главе 2, он возможен и в нерекурсивных решениях. В задаче, решаемой посредством последовательного разделения, обычно используется метод, содержащий два рекурсивных вызова самого себя (по одному для каждой половины задачи). При двоичном поиске используются два вызова, но фактически выполняется только один из них (какой именно — зависит от значения ключа). Алгоритм сортировки слиянием, рассмотренный позднее в этой главе, выполняет оба рекурсивных вызова (для сортировки двух половин массива).

Ханойская башня

Старая головоломка «Ханойская башня» состоит из дисков, насаженных на три стержня (рис. 6.10).

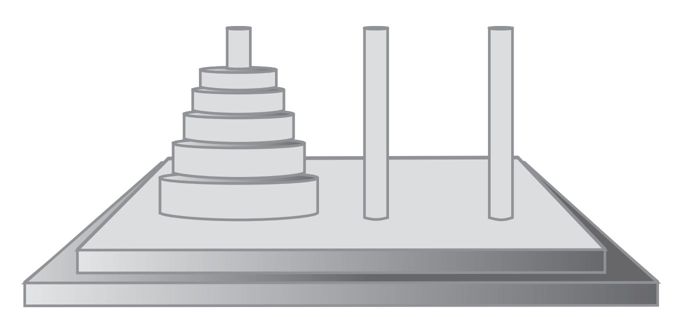


Рис. 6.10. Ханойская башня

Все диски имеют разный диаметр, а в середине у них просверлено отверстие, что позволяет складывать их в стопки. Изначально все диски находятся на стержне A. Цель головоломки — переложить все диски со стержня A на стержень C. Диски перекладываются по одному, причем запрещается класть диск на другой диск меньшего размера.

Древнее предание гласит, что где-то в Индии в забытом храме монахи трудятся день и ночь, перекладывая 64 золотых диска с одного алмазного стержня на другой. Когда их работа будет завершена, наступит конец света. Впрочем, ваше беспокойство рассеется, когда вы узнаете, сколько времени потребуется для перекладывания много меньшего количества дисков.

Приложение Towers Workshop

Запустите приложение Towers Workshop. Вы можете попытаться решить головоломку самостоятельно, перетаскивая мышью верхний диск из стопки на другой стержень. На рис. 6.11 показано, как выглядит головоломка после нескольких ходов.

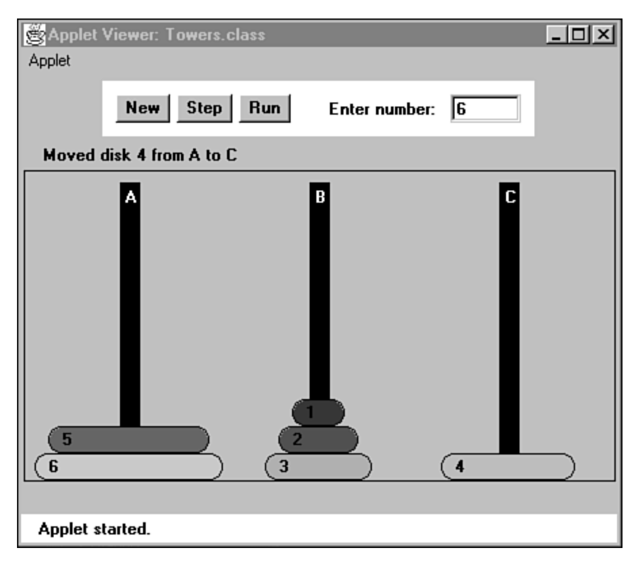


Рис. 6.11. Приложение Towers Workshop

С приложением Towers Workshop можно работать в трех режимах:

- Вы можете попробовать вручную решить головоломку, перетаскивая диски между стержнями.
- Многократно нажимайте кнопку Step и наблюдайте за тем, как алгоритм сам решает головоломку. На каждом шаге решения выводится сообщение с описанием текущей операции.
- При нажатии кнопки Run алгоритм решает головоломку без вашего участия; диски сами перемещаются между стержнями.

Чтобы начать решение заново, введите количество дисков (от 1 до 10) и дважды нажмите кнопку New. (После первого нажатия вам будет предложено подтвердить перезапуск.) Заданное количество дисков размещается на стержне A. Если перетащить диск мышью, вы уже не сможете воспользоваться кнопкой Step или Run; решение придется начинать заново кнопкой New. С другой стороны, в режиме пошагового или непрерывного решения можно переключиться на ручной режим, а также переключиться на пошаговый режим из непрерывного или наоборот.

Попробуйте решить головоломку вручную для небольшого количества дисков (скажем, 3 или 4). Поэкспериментируйте с более высокими числами. Приложение поможет на интуитивном уровне понять, как решается эта задача.

Перемещение поддеревьев

Назовем изначальную «пирамиду» из дисков на стержне A *деревом*. (Термин не имеет никакого отношения к древовидным структурам данных, которые будут рассматриваться позднее в книге.) В ходе экспериментов с приложением вы начнете замечать, что в процессе решения формируются стопки дисков меньшего размера. Назовем эти уменьшенные подобия дерева, содержащие лишь часть дисков, *поддеревьями*. Например, при перемещении четырех дисков можно заметить, что на одном из промежуточных шагов на стержне B строится поддерево из трех дисков (рис. 6.12).

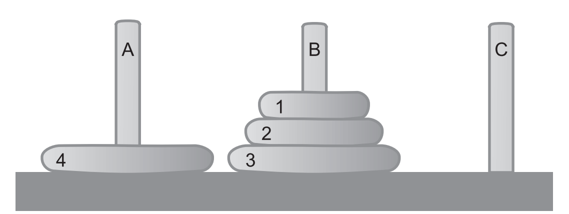


Рис. 6.12. Поддерево на стержне B

Такие поддеревья неоднократно образуются в ходе решения головоломки. Дело в том, что перемещение большего диска с одного стержня на другой возможно только с построением дерева: все меньшие диски должны быть размещены на промежуточном стержне, где они естественным образом формируют поддерево.

Следующее правило поможет вам в ручном решении головоломки. Если перемещаемое поддерево содержит нечетное количество дисков, начните с перемещения верхнего диска прямо на стержень, на котором должно быть построено поддерево. При перемещении поддерева с четным количеством дисков начните с перемещения верхнего диска на промежуточный стержень.

Рекурсивный алгоритм

Решение головоломки может быть представлено в рекурсивном виде с использованием поддеревьев. Предположим, требуется переместить все диски с исходного стержня (назовем его S) на приемный стержень (D). Также имеется промежуточный стержень (I). На стержне S находятся *n* дисков. Алгоритм перемещения выглядит так:

1. Переместить поддерево, состоящее из верхних $n - 1$ дисков, с S на I.
2. Переместить оставшийся (самый большой) диск с S на D.
3. Переместить поддерево с I на D.

В начале решения исходным является стержень A, промежуточным — B, а приемным — C. На рис. 6.13 показаны три этапа перемещения.

Сначала поддерево, состоящее из дисков 1, 2 и 3, перемещается на промежуточный стержень B. Затем самый большой диск 4 перемещается на стержень C, после чего поддерево перемещается с B на C.

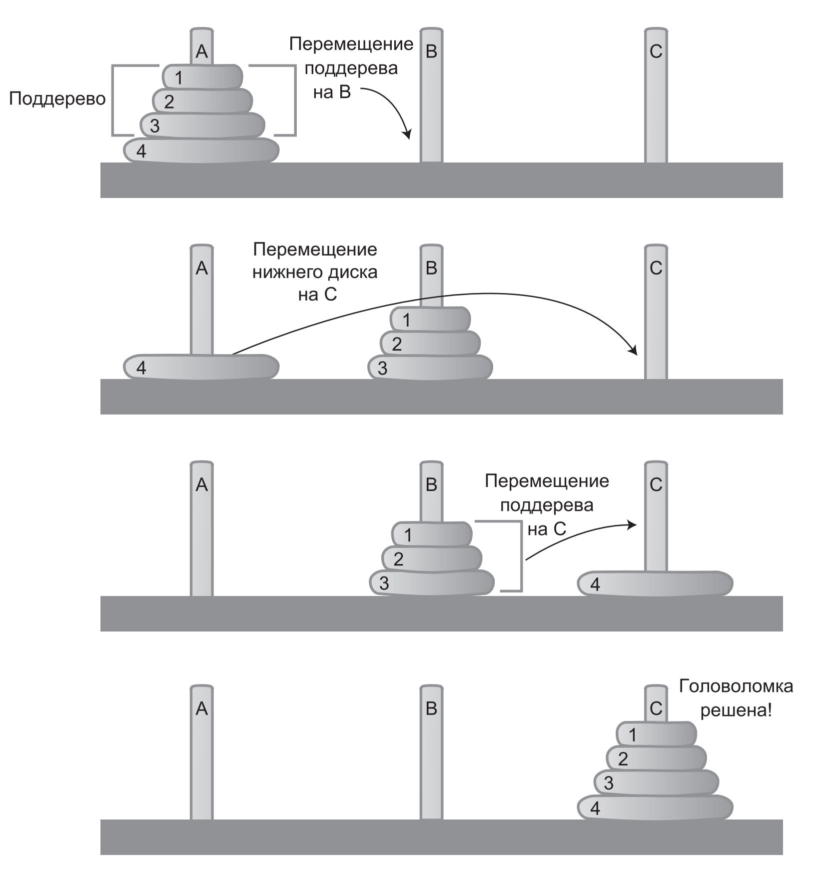


Рис. 6.13. Рекурсивное решение головоломки

Конечно, это решение ничего не говорит о том, как переместить поддерево из дисков 1, 2 и 3 на стержень B — ведь переместить все диски сразу невозможно, необходимо перекладывать их по одному. Задача перемещения поддерева из трех дисков не так тривиальна, и все же это проще, чем перемещать четыре диска.

Как выясняется, перемещение трех дисков со стержня A на стержень B выполняется за те же три шага, что и перемещение четырех дисков. Иначе говоря, поддерево из двух верхних дисков перемещается со стержня A на промежуточный стержень C, после чего диск 3 со стержня A перемещается на стержень B. Далее поддерево возвращается со стержня C на стержень B.

Как переместить поддерево из двух дисков с A на C? Сначала поддерево, состоящее из единственного диска (1), перемещается с A на B. Это базовое ограничение: один диск просто перекладывается со стержня на стержень, ничего другого делать не нужно. Затем больший диск (2) перемещается с A на C; остается лишь положить на него диск 1.

Программа towers.java

Программа towers.java решает головоломку «Ханойская башня» рекурсивным методом. Информация о ходе решения выводится по ходам; такой вывод программируется намного компактнее, чем вывод всего содержимого башен. Пользователь, просматривающий результаты выполнения, должен сам воспроизвести ходы, чтобы прийти к текущей позиции.

Программный код невероятно прост. Метод main() обращается с единственным вызовом к рекурсивному методу doTowers(). Последний продолжает рекурсивно вызывать себя до тех пор, пока головоломка не будет решена. В версии, представленной в листинге 6.4, изначально используются всего три диска, но вы можете перекомпилировать программу для любого количества дисков.

Листинг 6.4. Программа towers.java

```
// towers.java
// Решение головоломки "Ханойская башня"
// Запуск программы: C>java TowersApp
////////////////////////////////////////////////////////////////
class TowersApp
   {
   static int nDisks = 3;

   public static void main(String[] args)
      {
      doTowers(nDisks, 'A', 'B', 'C');
      }
//-----------------------------------------------------------
   public static void doTowers(int topN,
                            char from, char inter, char to)
      {
      if(topN==1)
         System.out.println("Disk 1 from " + from + " to "+ to);
      else
         {
         doTowers(topN-1, from, to, inter);  // from-->inter
         System.out.println("Disk " + topN +
         " from " + from + " to "+ to);
         doTowers(topN-1, inter, from, to);  // inter-->to
         }
      }
//-----------------------------------------------------------
   }  // Конец класса TowersApp
////////////////////////////////////////////////////////////////
```

Не забудьте, что три диска перемещаются со стержня A на стержень C. Результат выполнения программы:

```
Disk 1 from A to C
Disk 2 from A to B
Disk 1 from C to B
```

```
Disk 3 from A to C
Disk 1 from B to A
Disk 2 from B to C
Disk 1 from A to C
```

В аргументах doTowers() передается количество перемещаемых дисков, а также исходный (from), промежуточный (inter) и приемный (to) стержни. Количество дисков уменьшается на единицу при каждом рекурсивном вызове. Стержни тоже изменяются.

Далее приведен вывод программы с дополнительной разметкой, обозначающей точки входа и возврата из метода, аргументы и разновидность перемещения диска: базовое ограничение (Base case) для поддерева из единственного диска или последний нижний диск (Move bottom disk) после перемещения поддерева:

```
Enter (3 disks): s=A, i=B, d=C
   Enter (2 disks): s=A, i=C, d=B
      Enter (1 disk): s=A, i=B, d=C
         Base case: move disk 1 from A to C
      Return (1 disk)
      Move bottom disk 2 from A to B
      Enter (1 disk): s=C, i=A, d=B
         Base case: move disk 1 from C to B
      Return (1 disk)
   Return (2 disks)
   Move bottom disk 3 from A to C
   Enter (2 disks): s=B, i=A, d=C
      Enter (1 disk): s=B, i=C, d=A
         Base case: move disk 1 from B to A
      Return (1 disk)
      Move bottom disk 2 from B to C
      Enter (1 disk): s=A, i=B, d=C
         Base case: move disk 1 from A to C
      Return (1 disk)
   Return (2 disks)
Return (3 disks)
```

Проанализируйте эти результаты вместе с исходным кодом doTower(), и вам станет абсолютно ясно, как работает метод. Удивительно, как такой компактный код может решать столь нетривиальную (по крайней мере на первый взгляд) задачу.

Сортировка слиянием

Алгоритм сортировки слиянием завершает наше знакомство с рекурсией. Этот алгоритм сортировки намного эффективнее алгоритмов, представленных в главе 3, «Простая сортировка» — по крайней мере в отношении скорости. Если пузырьковая сортировка, сортировка методом вставки и сортировка методом выбора выполняются за время $O(N^2)$, то сортировка слиянием выполняется за время $O(N \times \log N)$. График на рис. 2.9 (глава 2) показывает, насколько ускоряется сортировка. Например, если N (количество сортируемых элементов) равно 10 000, то значение

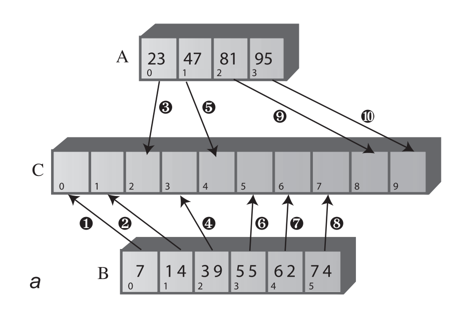
N^2 будет равно 100 000 000, а $N \times \log N$ — всего 40 000. Если сортировка слиянием для такого количества элементов займет 40 секунд, то сортировка методом вставок потребует около 28 часов. Кроме того, сортировка слиянием относительно легко реализуется. На концептуальном уровне она проще алгоритмов быстрой сортировки и сортировки Шелла, описанных в следующей главе.

Недостаток сортировки слиянием заключается в том, что она требует выделения в памяти дополнительного массива с размером, равным размеру сортируемого массива. Если исходный массив с трудом помещается в памяти, то сортировка слиянием вам не подойдет, но при наличии достаточного свободного пространства этот алгоритм весьма эффективен.

Слияние двух отсортированных массивов

Центральное место в алгоритме сортировки слиянием занимает слияние двух предварительно отсортированных массивов. В результате слияния двух сортированных массивов A и B создается третий массив C, который содержит все элементы A и B, также расположенные в порядке сортировки. Сначала мы рассмотрим сам процесс слияния, а потом разберемся, как он используется при сортировке.

Представьте, что у вас имеются два предварительно отсортированных массива (причем необязательно одинакового размера). Допустим, массив A содержит 4 элемента, а массив B — 6. Массивы объединяются посредством слияния в массив C, изначально состоящий из 10 пустых ячеек (рис. 6.14).



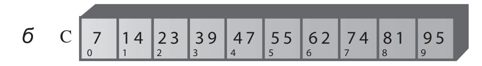


Рис. 6.14. Слияние двух массивов: *а* — до слияния; *б* — после слияния

На рисунке числа в кружках обозначают порядок перемещения элементов из A и B в C. В таблице 6.3 перечислены сравнения, необходимые для определения копируемых элементов. Нумерация операций в таблице соответствует нумерации на рисунке. После каждого сравнения меньший элемент копируется в A.

Так как после выполнения шага 8 массив B не содержит элементов, дополнительные сравнения не нужны; все оставшиеся элементы просто копируются из A в C.

Таблица 6.3. Операции слияния

Шаг	Сравнение	Копирование
1	23 и 7	7 из B в C
2	23 и 14	14 из A в C
3	23 и 39	23 из A в C
4	39 и 47	39 из B в C
5	55 и 47	47 из A в C
6	55 и 81	55 из B в C
7	62 и 81	62 из B в C
8	74 и 81	74 из B в C
9		81 из A в C
10		95 из A в C

В листинге 6.5 приведена программа на языке Java, которая выполняет сортировку слиянием для массивов на рис. 6.14 и табл. 6.3. Эта программа не является рекурсивной; она всего лишь помогает разобраться в сортировке слиянием.

Листинг 6.5. Программа merge.java

```
// merge.java
// Слияние двух массивов
// Запуск программы: C>java MergeApp
////////////////////////////////////////////////////////////////
class MergeApp
   {
   public static void main(String[] args)
      {
      int[] arrayA = {23, 47, 81, 95};
      int[] arrayB = {7, 14, 39, 55, 62, 74};
      int[] arrayC = new int[10];

      merge(arrayA, 4, arrayB, 6, arrayC);
      display(arrayC, 10);
      }
//-----------------------------------------------------------
                                  // Слияние массивов A и B в массив C
   public static void merge( int[] arrayA, int sizeA,
                             int[] arrayB, int sizeB,
                             int[] arrayC )
      {
      int aDex=0, bDex=0, cDex=0;
      while(aDex < sizeA && bDex < sizeB)  // Ни один из массивов не пуст
         if( arrayA[aDex] < arrayB[bDex] )
            arrayC[cDex++] = arrayA[aDex++];
```

продолжение ⇨

Листинг 6.5 (*продолжение*)

```
        else
           arrayC[cDex++] = arrayB[bDex++];
     while(aDex < sizeA)                   // Массив arrayB пуст,
        arrayC[cDex++] = arrayA[aDex++];  // в arrayA остались элементы
     while(bDex < sizeB)                   // Массив arrayA пуст,
        arrayC[cDex++] = arrayB[bDex++];  // в arrayB остались элементы
     }
//-----------------------------------------------------------
                                           // Вывод содержимого массива
   public static void display(int[] theArray, int size)
     {
     for(int j=0; j<size; j++)
        System.out.print(theArray[j] + " ");
     System.out.println("");
     }
//-----------------------------------------------------------
   }  // Конец класса MergeApp
////////////////////////////////////////////////////////////////
```

Метод main() создает массивы arrayA, arrayB и arrayC. Затем вызывается метод merge(), осуществляющий слияние arrayA и arrayB в arrayC, и программа выводит полученное содержимое arrayC. Результат выполнения:

```
7 14 23 39 47 55 62 74 81 95
```

Метод merge() содержит три цикла while. Первый цикл перебирает массивы arrayA и arrayB, сравнивает элементы и копирует меньший из них в arrayC.

Второй цикл используется в ситуации, в которой из массива arrayB уже были извлечены все элементы, а в массиве arrayA элементы все еще остаются (именно это происходит в нашем примере, где в arrayA оставались элементы 81 и 95). Цикл просто копирует оставшиеся элементы из arrayA в arrayC.

Третий цикл решает аналогичную задачу, когда все элементы были извлечены из массива arrayA, а массив arrayB еще не пуст; оставшиеся элементы копируются в arrayC.

Сортировка слиянием

Идея сортировки слиянием заключается в том, чтобы разделить массив пополам, отсортировать каждую половину, а затем воспользоваться методом merge() для слияния двух половин в один отсортированный массив. Как отсортировать каждую половину? Глава посвящена рекурсии, поэтому, вероятно, ответ вам уже известен: половина делится на две четверти, каждая четверть сортируется по отдельности, после чего две четверти объединяются в отсортированную половину.

Аналогичным образом каждая пара 8-х частей сливается в отсортированную четверть и т. д. Массив делится снова и снова, пока не будет получен подмассив из единственного элемента. Это базовое ограничение; предполагается, что массив из одного элемента уже отсортирован.

Мы видели, что в общем случае каждый вызов рекурсивного метода приводит к уменьшению некоторого параметра, а при каждом возвращении из метода этот параметр снова увеличивается. В mergeSort() при каждом рекурсивном вызове диа-

пазон делится надвое, а при каждом возвращении — две половины объединяются в один больший диапазон.

Когда метод `mergeSort()` возвращает управление после обнаружения двух массивов, содержащих по одному элементу, он объединяет их в отсортированный массив из двух элементов. Далее каждая пара полученных 2-элементных массивов объединяется в массив из 4-х элементов. Процесс продолжается с массивами постоянно увеличивающихся размеров до тех пор, пока весь массив не будет отсортирован. За сортировкой проще всего наблюдать для массивов, исходный размер которых составляет степень 2, как показано на рис. 6.15.

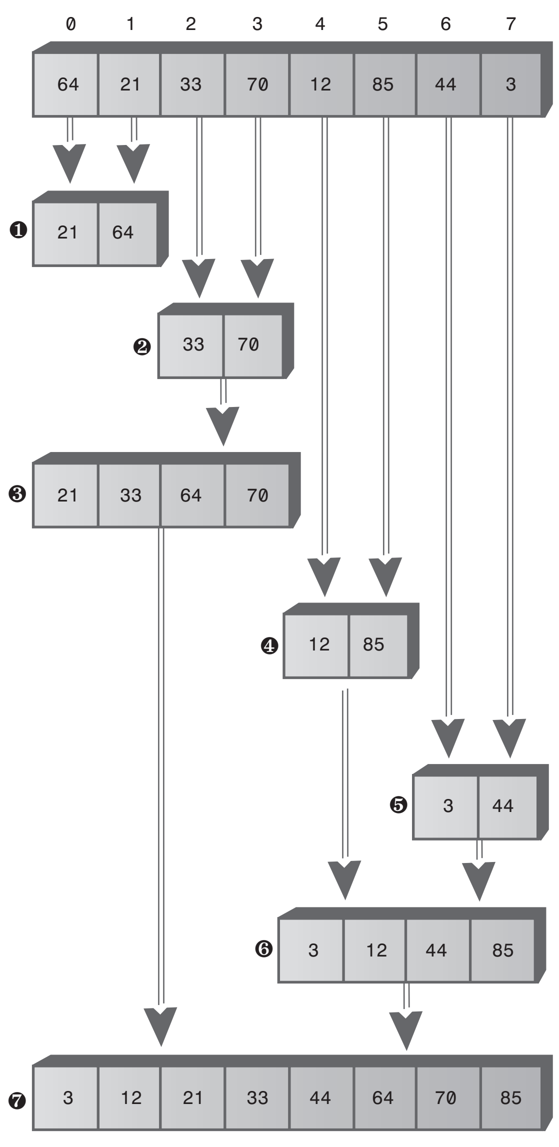


Рис. 6.15. Последовательное слияние с увеличением размера массивов

Сначала в нижней половине массива диапазоны 0-0 и 1-1 объединяются в диапазон 0-1. Конечно, 0-0 и 1-1 не являются полноценными диапазонами; они состоят лишь из одного элемента, то есть относятся к базовым ограничениям. Аналогичным образом диапазоны 2-2 и 3-3 объединяются в 2-3, после чего диапазоны 0-1 и 2-3 объединяются в 0-3.

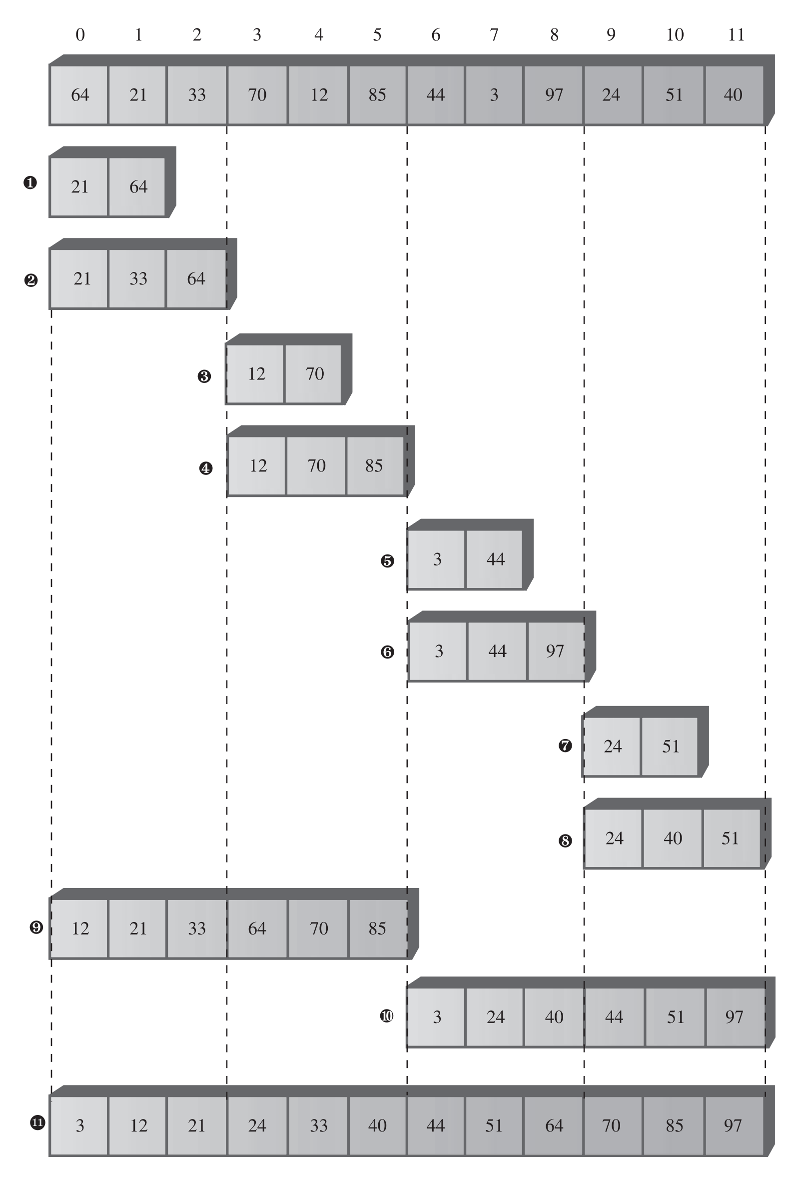


Рис. 6.16. Размер массива не является степенью 2

В верхней половине массива диапазоны 4-4 и 5-5 объединяются в 4-5, диапазоны 6-6 и 7-7 объединяются в 6-7, а диапазоны 4-5 и 6-7 объединяются в 4-7. Наконец, верхняя половина 0-3 и нижняя половина 4-7 объединяются в полный массив 0-7, элементы которого отсортированы.

Если размер массива не является степенью 2, в слиянии участвуют массивы разных размеров. Например, на рис. 6.16 показано, как выполняется сортировка массива из 12 элементов. Здесь массив из двух элементов объединяется с массивом из одного элемента, образуя массив из трех элементов.

Сначала «одноэлементные» диапазоны 0-0 и 1-1 объединяются в диапазон 0-1 из двух элементов. Затем диапазон 0-1 объединяется с одноэлементным диапазоном 2-2, образуя состоящий из трех элементов диапазон 0-2. Он объединяется с диапазоном 3-5, также состоящим из трех элементов. Процесс продолжается до тех пор, пока весь массив не будет отсортирован.

Обратите внимание: при сортировке слиянием два разных массива не объединяются в третий, как это делалось в программе `merge.java`. Вместо этого осуществляется слияние частей одного массива внутри него самого.

Резонно спросить, где все эти подмассивы хранятся в памяти? Алгоритм создает рабочую область, размер которой равен размеру исходного массива. Таким образом, подмассивы из исходного массива копируются в соответствующие места рабочей области. После каждого слияния данные из рабочей области копируются в исходный массив.

Приложение MergeSort Workshop

Процесс сортировки становится более понятным, когда вы видите его собственными глазами. Запустите приложение MergeSort Workshop. Повторные нажатия кнопки `Step` приводят к пошаговому выполнению сортировки слиянием. На рис. 6.17 показано, как выглядит приложение после первых трех нажатий.

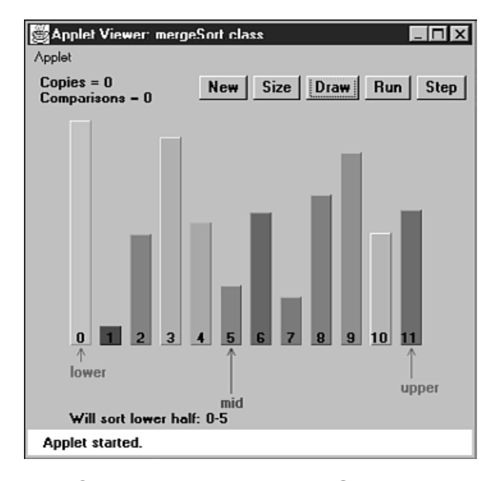


Рис. 6.17. Приложение MergeSort Workshop

Стрелки Lower и Upper отмечают границы диапазона, обрабатываемого алгоритмом в настоящий момент, а стрелка Mid отмечает середину диапазона. В исходном состоянии диапазон занимает весь массив, а затем уменьшается вдвое при каждом рекурсивном вызове mergeSort(). Если диапазон состоит из одного элемента, mergeSort() возвращает управление немедленно (базовое ограничение). В противном случае происходит слияние двух подмассивов. Приложение выводит сообщения с описанием выполняемых действий и диапазонов, с которыми они выполняются (например, Entering mergeSort: 0-5).

На многих этапах сортировки метод mergeSort() вызывает сам себя или возвращает управление. Сравнение и копирование выполняется только во время процесса слияния, когда в приложении выводятся сообщения вида Merged 0-0 and 1-1. Как происходит слияние, вы не видите, потому что рабочая область в приложении не показана. Однако результат становится видимым при копировании соответствующей части рабочей области обратно в исходный (видимый) массив: столбцы указанного диапазона отображаются в отсортированном порядке.

Сначала сортируются первые два столбца, затем первые три, потом два столбца в диапазоне 3-4, три столбца в диапазоне 3-5, шесть столбцов в диапазоне 0-5 и так далее в соответствии с последовательностью на рис. 6.16. В конечном итоге будут отсортированы все столбцы.

Кнопка Run запускает алгоритм на непрерывное выполнение. Процесс можно в любой момент прервать кнопкой Step; выполните столько шагов сортировки, сколько потребуется, и продолжите сортировку в непрерывном режиме повторным нажатием Run.

Как и во всех остальных приложениях Workshop, кнопка New инициализирует массив новой группой несортированных столбцов и позволяет выбрать между случайным расположением и сортировкой в обратном порядке. Кнопка Size предназначена для выбора размера массива (12 или 100 элементов).

Особенно поучительно проследить за тем, как алгоритм работает над сортировкой 100 столбцов, отсортированных в обратном порядке. В процессе сортировки хорошо видно, как две половины сортируются по отдельности, а затем сливаются в единое целое и как диапазоны постепенно увеличиваются.

Программа mergeSort.java

Полный код программы mergeSort.java будет рассмотрен ниже. А пока сосредоточимся на методе, непосредственно выполняющем сортировку слиянием. Он выглядит так:

```
private void recMergeSort(long[] workSpace, int lowerBound,
                                             int upperBound)
   {
   if(lowerBound == upperBound)            // Если только один элемент,
      return;                              // сортировка не требуется.
   else
      {                                    // Поиск середины
      int mid = (lowerBound+upperBound) / 2;
```

```
                                     // Сортировка нижней половины
      recMergeSort(workSpace, lowerBound, mid);
                                     // Сортировка верхней половины
      recMergeSort(workSpace, mid+1, upperBound);
                                     // Слияние
      merge(workSpace, lowerBound, mid+1, upperBound);
      }
   }
```

Как видите, кроме проверки базового ограничения, этот метод содержит всего четыре команды. Одна команда вычисляет среднюю точку диапазона, затем следуют два рекурсивных вызова recMergeSort() (по одному для каждой половины массива) и, наконец, вызов merge() объединяет две отсортированные половины. Базовое ограничение заключается в том, что диапазон состоит из единственного элемента (lowerBound==upperBound); в этом случае происходит немедленный возврат.

В программе mergeSort.java пользователю класса виден только метод mergeSort(). Он создает массив workSpace[] и вызывает рекурсивную функцию recMergeSort() для фактического выполнения сортировки. Массив рабочей области создается в mergeSort(), так как при его создании в recMergeSort() он заново создавался бы с каждым рекурсивным вызовом (конечно, это было бы крайне неэффективно). Метод merge() в предыдущей версии программы merge.java (листинг 6.5) использовал три разных массива: два исходных и один приемный. Метод merge() в программе mergeSort.java использовал всего один массив: поле theArray класса DArray. В аргументах метода merge() передается начальная точка подмассива нижней половины, начальная точка подмассива верхней половины, а также верхняя граница подмассива верхней половины. На основании этой информации метод вычисляет размеры подмассивов.

В листинге 6.6 приведен полный код программы mergeSort.java, которая использует разновидность класса массива из главы 2 — класс DArray дополняется методами mergeSort() и recMergeSort(). Метод main() создает массив, вставляет в него 12 элементов, выводит содержимое массива, сортирует элементы методом mergeSort() и снова выводит содержимое массива.

Листинг 6.6. Программа mergeSoft.java

```
// mergeSort.java
// Рекурсивная реализация сортировки слиянием
// Запуск программы: C>java MergeSortApp
////////////////////////////////////////////////////////////////
class DArray
   {
   private long[] theArray;          // Ссылка на массив theArray
   private int nElems;               // Количество элементов данных
//-----------------------------------------------------------
   public DArray(int max)            // Конструктор
      {
      theArray = new long[max];      // Создание массива
      nElems = 0;
      }
```

продолжение ⇨

Листинг 6.6 (*продолжение*)

```
//-------------------------------------------------------------
   public void insert(long value)    // Занесение элемента в массив
      {
      theArray[nElems] = value;      // Вставка элемента
      nElems++;                      // Увеличение размера
      }
//-------------------------------------------------------------
   public void display()             // Вывод содержимого массива
      {
      for(int j=0; j<nElems; j++)    // Для каждого элемента
      System.out.print(theArray[j] + " ");  // Вывод
      System.out.println("");
      }
//-------------------------------------------------------------
   public void mergeSort()           // Вызывается из main()
      {                              // Рабочая область
      long[] workSpace = new long[nElems];
      recMergeSort(workSpace, 0, nElems-1);
      }
//-------------------------------------------------------------
   private void recMergeSort(long[] workSpace, int lowerBound,
                                               int upperBound)
      {
      if(lowerBound == upperBound)            // Если только один элемент,
         return;                              // сортировка не требуется.
      else
         {                                    // Поиск середины
         int mid = (lowerBound+upperBound) / 2;
                                              // Сортировка нижней половины
         recMergeSort(workSpace, lowerBound, mid);
                                              // Сортировка верхней половины
         recMergeSort(workSpace, mid+1, upperBound);
                                              // Слияние
         merge(workSpace, lowerBound, mid+1, upperBound);
         }
      }
//-------------------------------------------------------------
   private void merge(long[] workSpace, int lowPtr,
                                 int highPtr, int upperBound)
      {
      int j = 0;                             // Индекс в рабочей области
      int lowerBound = lowPtr;
      int mid = highPtr-1;
      int n = upperBound-lowerBound+1;       // Количество элементов

      while(lowPtr <= mid && highPtr <= upperBound)
         if( theArray[lowPtr] < theArray[highPtr] )
            workSpace[j++] = theArray[lowPtr++];
         else
```

```
            workSpace[j++] = theArray[highPtr++];

      while(lowPtr <= mid)
         workSpace[j++] = theArray[lowPtr++];

      while(highPtr <= upperBound)
         workSpace[j++] = theArray[highPtr++];

      for(j=0; j<n; j++)
         theArray[lowerBound+j] = workSpace[j];
      }
//-------------------------------------------------------------
   }  // Конец класса DArray
////////////////////////////////////////////////////////////////
class MergeSortApp
   {
   public static void main(String[] args)
         {
         int maxSize = 100;             // Размер массива
         DArray arr;                    // Ссылка на массив
         arr = new DArray(maxSize);     // Создание массива

         arr.insert(64);                // Вставка элементов
         arr.insert(21);
         arr.insert(33);
         arr.insert(70);
         arr.insert(12);
         arr.insert(85);
         arr.insert(44);
         arr.insert(3);
         arr.insert(99);
         arr.insert(0);
         arr.insert(108);
         arr.insert(36);

         arr.display();                 // Вывод содержимого массива

         arr.mergeSort();               // Сортировка слиянием

         arr.display();                 // Повторный вывод
         }
      }  // Конец класса MergeSortApp
////////////////////////////////////////////////////////////////
```

Программа выводит два массива, до и после сортировки:

```
64 21 33 70 12 85 44 3 99 0 108 36
0 3 12 21 33 36 44 64 70 85 99 108
```

Включение вспомогательных команд в метод recMergeSort() позволит сгенерировать дополнительные комментарии, поясняющие смысл выполняемых действий. Следующий пример показывает, как могут выглядеть такие комментарии

для массива из четырех элементов {64, 21, 33, 70} (то есть для нижней половины массива на рис. 6.15).

```
Entering 0-3
   Will sort low half of 0-3
   Entering 0-1
      Will sort low half of 0-1
         Entering 0-0
         Base-Case Return 0-0
      Will sort high half of 0-1
         Entering 1-1
         Base-Case Return 1-1
      Will merge halves into 0-1
   Return 0-1                        theArray=21 64 33 70
   Will sort high half of 0-3
   Entering 2-3
      Will sort low half of 2-3
         Entering 2-2
         Base-Case Return 2-2
      Will sort high half of 2-3
         Entering 3-3
         Base-Case Return 3-3
      Will merge halves into 2-3
   Return 2-3                        theArray=21 64 33 70
   Will merge halves into 0-3
Return 0-3                           theArray=21 33 64 70
```

Примерно такой же результат был бы сгенерирован приложением MergeSort Workshop при сортировке четырех элементов. Проанализируйте эти данные; сравнение с кодом `recMergeSort()` и рис. 6.15 прояснит подробности процесса сортировки.

Эффективность сортировки слиянием

Как упоминалось ранее, сортировка слиянием выполняется за время $O(N \times \log N)$. Откуда это известно? Давайте посмотрим, как вычисляется количество операций копирования и сравнения элемента данных с другими элементами в ходе работы этого алгоритма. Предполагается, что копирование и сравнение занимают больше всего времени, а рекурсивные вызовы и возвраты не добавляют сколько-нибудь заметных затрат.

Количество операций копирования

Взгляните на рис. 6.15. Каждая ячейка под верхней линией представляет элемент, копируемый из массива в рабочую область.

Подсчет всех ячеек на рис. 6.15 (семь пронумерованных шагов) показывает, что для сортировки 8 элементов требуется 24 операции копирования. Значение $\log_2 8$ равно 3, а значение $8 \times \log_2 8$ равно 24. Таким образом, для 8 элементов количество операций копирования пропорционально $N \times \log_2 N$.

К вычислению также можно подойти с другой стороны: для сортировки восьми элементов требуются три уровня, на каждом из которых выполняется 8 операций копирования. Под «уровнем» здесь понимаются все операции копирования в подмассивы одного размера. На первом уровне выделяются четыре 2-элементных подмассива; на втором — два 4-элементных; на третьем — один 8-элементный. На каждом уровне используются 8 элементов; снова получается $3 \times 8 = 24$ операции копирования.

На рис. 6.15, рассматривая только половину диаграммы, мы видим, что для массива из четырех элементов (шаги 1, 2 и 3) требуется 8 операций копирования, а для массива из двух элементов — 2 операции копирования. Аналогичные вычисления позволяют определить количество операций копирования для массивов большего размера. В таблице 6.4 приведена краткая сводка количества операций для разных размеров массива.

Таблица 6.4. Количество операций для N, равных степени 2

N	$\log_2 N$	Количество операций копирования в рабочей области ($N \times \log_2 N$)	Общее количество операций копирования	Количество сравнений макс (мин)
2	1	2	4	1 (1)
4	2	8	16	5 (4)
8	3	24	48	17 (12)
16	4	64	128	49 (32)
32	5	160	320	129 (80)
64	6	384	768	321 (192)
128	7	896	1792	769 (448)

Элементы копируются не только в рабочую область, но и обратно в исходный массив. Таким образом, количество операций копирования удваивается (столбец «Общее количество операций копирования»). В последнем столбце табл. 6.4 приведены данные о количестве сравнений; вскоре мы вернемся к ним.

Если N не является степенью 2, вычисление количества операций копирования и сравнения немного усложняется, но результаты лежат между соответствующими значениями для степеней 2. Для 12 элементов общее количество операций копирования будет равно 88, а для 100 элементов — 1344.

Количество операций сравнения

В алгоритме сортировки слиянием количество сравнений всегда немного меньше количества операций копирования. Насколько меньше? Если предположить, что размер массива является степенью 2, то для каждой операции слияния максимальное количество сравнений всегда на единицу меньше количества элементов, участвующих в слиянии, а минимальное равно половине их количества. Почему это именно так, видно из рис. 6.18, на котором представлены два возможных случая слияния двух массивов, содержащих по 4 элемента.

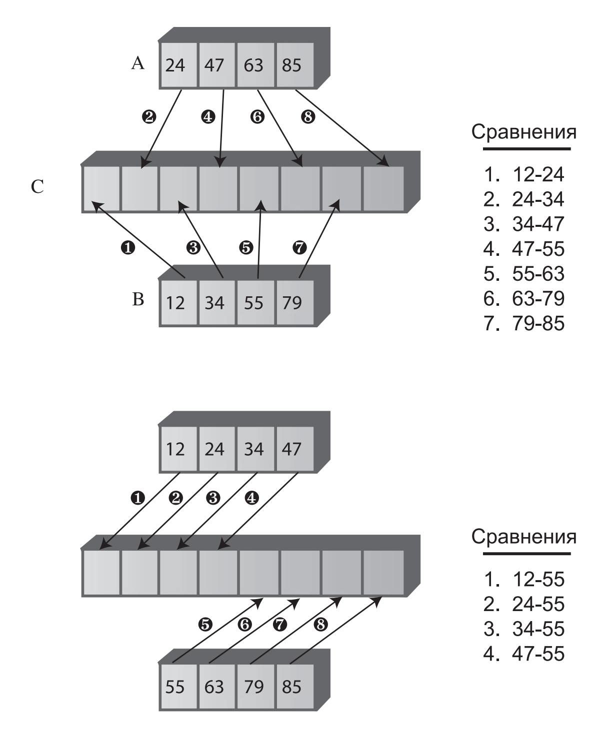


Рис. 6.18. Максимальное и минимальное количество сравнений

В первом случае элементы чередуются и для их слияния придется выполнить семь сравнений. Во втором случае все элементы одного массива меньше всех элементов другого, поэтому достаточно всего четырех сравнений.

В каждой сортировке задействовано много слияний, поэтому мы должны просуммировать сравнения по всем слияниям. Возвращаясь к рис. 6.15, мы видим, что для сортировки восьми элементов требуется семь операций слияния. Сводка количества объединяемых элементов и сравнений представлена в табл. 6.5.

Таблица 6.5. Сравнения при сортировке восьми элементов

Шаг	1	2	3	4	5	6	7	Итого
Количество элементов при слиянии (N)	2	2	4	2	2	4	8	24
Максимальное количество сравнений (N–1)	1	1	3	1	1	3	7	17
Минимальное количество сравнений (N/2)	1	1	2	1	1	2	4	12

При каждом слиянии максимальное количество сравнений на единицу меньше количества элементов. Суммирование значений для всех операций слияния дает в сумме 17.

Минимальное количество сравнений всегда равно половине количества элементов при слиянии. Суммируя по всем слияниям, мы получаем 12. Значения в по-

следнем столбце табл. 6.4 вычислялись именно таким образом. Точное количество сравнений при сортировке конкретного массива зависит от расположения данных, но оно заведомо находится где-то в диапазоне между минимальным и максимальным значениями.

Устранение рекурсии

Одни алгоритмы хорошо адаптируются к рекурсии, другие — нет. Как мы уже видели, рекурсивные методы `triangle()` и `factorial()` более эффективно реализуются с использованием простого цикла. С другой стороны, различные алгоритмы последовательного разделения (такие, как сортировка слиянием) очень хорошо работают в рекурсивной форме.

Часто алгоритм легко представляется рекурсивным методом на концептуальном уровне, но на практике рекурсия оказывается недостаточно эффективной. В таких случаях от рекурсии стоит отказаться. Преобразования такого рода часто основаны на использовании стека.

Рекурсия и стеки

Между рекурсией и стеками существует тесная связь. Компиляторы чаще всего применяют стеки для реализации рекурсии. Как уже говорилось ранее, при вызове метода компилятор заносит аргументы метода и адрес возврата (по которому должно быть передано управление при выходе из метода) в стек, после чего передает управление методу. При выходе из метода значения извлекаются из стека. Аргументы игнорируются, а управление передается по адресу возврата.

Моделирование рекурсивного метода

В этом разделе мы покажем, как любое рекурсивное решение преобразуется в решение на базе стека. Помните рекурсивный метод `triangle()` из первого раздела этой главы?

```
int triangle(int n)
   {
   if(n==1)
      return 1;
   else
      return( n + triangle(n-1) );
   }
```

Мы разобьем этот алгоритм на отдельные операции, каждая из которых станет отдельной секцией `case` в конструкции `switch`. (В C++ и некоторых других языках аналогичная декомпозиция может быть реализована командами `goto`, но в Java `goto` не поддерживается.)

Команда `switch` заключается в метод с именем `step`. При каждом вызове `step()` выполняется одна секция `case`. Многократный вызов `step()` в конечном итоге приведет к выполнению всего кода алгоритма.

Приведенный выше метод triangle() выполняет операции двух видов. Во-первых, он выполняет арифметические действия, необходимые для вычисления треугольных чисел: сравнение n с 1 и суммирование n с результатом предыдущих рекурсивных вызовов. Однако triangle() также выполняет операции, необходимые для управления самим методом, включая передачу управления, обращение к аргументам и адресу возврата. Эти операции не видны в программном коде, они встраиваются во все методы. В общих чертах при вызове метода происходит следующее:

- Аргументы и адрес возврата заносятся в стек.
- Метод обращается к своим аргументам, читая значения с вершины стека.
- Непосредственно перед возвращением управления метод читает из стека адрес возврата, после чего извлекает этот адрес и свои аргументы из стека и уничтожает их.

Программа stackTriangle.java содержит три класса: Params, StackX и StackTriangleApp. Класс Params инкапсулирует адрес возврата и аргумент метода n; объекты этого класса заносятся в стек. Класс StackX аналогичен классам стека из других глав, но хранятся в нем объекты класса Params. Класс StackTriangleApp содержит четыре метода: main(), recTriangle(), step() и стандартный метод getInt() для ввода числовых данных.

Метод main() запрашивает у пользователя число, вызывает метод recTriangle() для вычисления треугольного числа с номером n, после чего выводит результат.

Метод recTriangle() создает объект StackX и инициализирует codePart значением 1. Далее начинается цикл while с многократным вызовом step(). Выполнение цикла прерывается лишь после того, как step() вернет true при достижении case 6 (точка выхода). По сути, метод step() представляет собой одну большую конструкцию switch, в которой каждая секция case соответствует фрагменту кода исходного метода triangle(). Полный код программы stackTriangle.java приведен в листинге 6.7.

Листинг 6.7. Программа stackTriangle.java

```
// stackTriangle.java
// Вычисление треугольных чисел с заменой рекурсии стеком
// Запуск программы: C>java StackTriangleApp
import java.io.*;                 // Для ввода/вывода
////////////////////////////////////////////////////////////////
class Params     // Параметры, сохраняемые в стеке
   {
   public int n;
   public int returnAddress;

   public Params(int nn, int ra)
      {
      n=nn;
      returnAddress=ra;
      }
   }  // Конец класса Params
////////////////////////////////////////////////////////////////
```

```
class StackX
   {
   private int maxSize;         // Размер массива StackX
   private Params[] stackArray;
   private int top;             // Вершина стека
//-----------------------------------------------------------------
   public StackX(int s)         // Конструктор
      {
      maxSize = s;              // Определение размера массива
      stackArray = new Params[maxSize];  // Создание массива
      top = -1;                 // Массив пока не содержит элементов
      }
//-----------------------------------------------------------------
   public void push(Params p)   // Размещение элемента на вершине стека
      {
      stackArray[++top] = p;    // Увеличение top, вставка элемента
      }
//-----------------------------------------------------------------
   public Params pop()          // Извлечение элемента с вершины стека
      {
      return stackArray[top--]; // Обращение к элементу, уменьшение top
      }
//-----------------------------------------------------------------
   public Params peek()         // Чтение элемента на вершине стека
      {
      return stackArray[top];
      }
//-----------------------------------------------------------------
   }  // Конец класса StackX
////////////////////////////////////////////////////////////////
class StackTriangleApp
   {
   static int theNumber;
   static int theAnswer;
   static StackX theStack;
   static int codePart;
   static Params theseParams;
//----------------------------------------------------------------
   public static void main(String[] args) throws IOException
      {
      System.out.print("Enter a number: ");
      theNumber = getInt();
      recTriangle();
      System.out.println("Triangle="+theAnswer);
      }
//----------------------------------------------------------------
   public static void recTriangle()
      {
      theStack = new StackX(10000);
```

продолжение ⇨

Листинг 6.6 (*продолжение*)

```
      codePart = 1;
      while( step() == false)  // Вызывать, пока step() не вернет true
         ;                     // Пустое тело цикла
      }
//-------------------------------------------------------------
   public static boolean step()
      {
      switch(codePart)
         {
         case 1:                              // Исходный вызов
            theseParams = new Params(theNumber, 6);
            theStack.push(theseParams);
            codePart = 2;
            break;
         case 2:                              // Вход в метод
            theseParams = theStack.peek();
            if(theseParams.n == 1)            // Проверка
               {
               theAnswer = 1;
               codePart = 5;                  // Выход
               }
            else
               codePart = 3;                  // Рекурсивный вызов
            break;
         case 3:                              // Вызов метода
            Params newParams = new Params(theseParams.n - 1, 4);
            theStack.push(newParams);
            codePart = 2;  // Вход в метод
            break;
         case 4:                              // Вычисление
            theseParams = theStack.peek();
            theAnswer = theAnswer + theseParams.n;
            codePart = 5;
            break;
         case 5:                              // Выход из метода
            theseParams = theStack.peek();
            codePart = theseParams.returnAddress; // (4 или 6)
            theStack.pop();
            break;
         case 6:                              // Точка возврата
            return true;
         }
      return false;
      }
//-------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
```

```
      String s = br.readLine();
      return s;
      }
//-------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
//--------------------------------------------------------------
   }  // Конец класса StackTriangleApp
//////////////////////////////////////////////////////////////
```

Программа вычисляет треугольные числа точно так же, как это делала программа triangle.java (см. листинг 6.1). Пример:

```
Enter a number: 100
Triangle=5050
```

На рис. 6.19 показано, какие секции case соответствуют тем или иным частям алгоритма.

Программа имитирует вызов метода, но имитация, в отличие от полноценного метода Java, не обладает именем. Для удобства назовем имитируемый метод simMeth(). Исходный вызов simMeth() (case 1) заносит в стек значение, введенное пользователем, и возвращаемое значение 6 и переходит к точке входа simMeth() (case 2).

На входе (case 2) simMeth() проверяет, равен ли его аргумент 1. Для обращения к аргументу используется чтение с вершины стека. Если аргумент равен 1, срабатывает базовое ограничение, и управление передается на выход simMeth() (case 5). В противном случае метод рекурсивно вызывает себя (case 3).

Рекурсивный вызов состоит из занесения n-1 и адреса возврата 4 в стек и передачи управления на вход метода (case 2).

При возвращении из рекурсивного вызова simMeth() прибавляет свой аргумент n к значению, полученному в результате вызова, после чего возвращает управление (case 5). При выходе из стека извлекается последний объект Params; содержащаяся в нем информация уже не нужна.

Адрес возврата, указанный при исходном вызове, был равен 6, поэтому при выходе из метода управление передается в секцию case 6. Этот код возвращает true, чтобы цикл while в recTriangle() узнал о завершении цикла.

Обратите внимание: в описании выполняемых simMeth() действий используются такие термины, как «аргумент», «рекурсивный вызов» и «адрес возврата»; они обозначают модели указанных понятий, а не их обычные реализации в Java.

Если добавить в каждую секцию case дополнительные команды вывода, описывающие выполняемые действия, можно получить подробный листинг следующего вида:

```
Enter a number: 4
case 1. theAnswer=0  Stack:
case 2. theAnswer=0  Stack: (4, 6)
case 3. theAnswer=0  Stack: (4, 6)
case 2. theAnswer=0  Stack: (4, 6) (3, 4)
```

```
case 3. theAnswer=0  Stack: (4, 6) (3, 4)
case 2. theAnswer=0  Stack: (4, 6) (3, 4) (2, 4)
case 3. theAnswer=0  Stack: (4, 6) (3, 4) (2, 4)
case 2. theAnswer=0  Stack: (4, 6) (3, 4) (2, 4) (1, 4)
case 5. theAnswer=1  Stack: (4, 6) (3, 4) (2, 4) (1, 4)
case 4. theAnswer=1  Stack: (4, 6) (3, 4) (2, 4)
case 5. theAnswer=3  Stack: (4, 6) (3, 4) (2, 4)
case 4. theAnswer=3  Stack: (4, 6) (3, 4)
case 5. theAnswer=6  Stack: (4, 6) (3, 4)
case 4. theAnswer=6  Stack: (4, 6)
case 5. theAnswer=10 Stack: (4, 6)
case 6. theAnswer=10 Stack:
Triangle=10
```

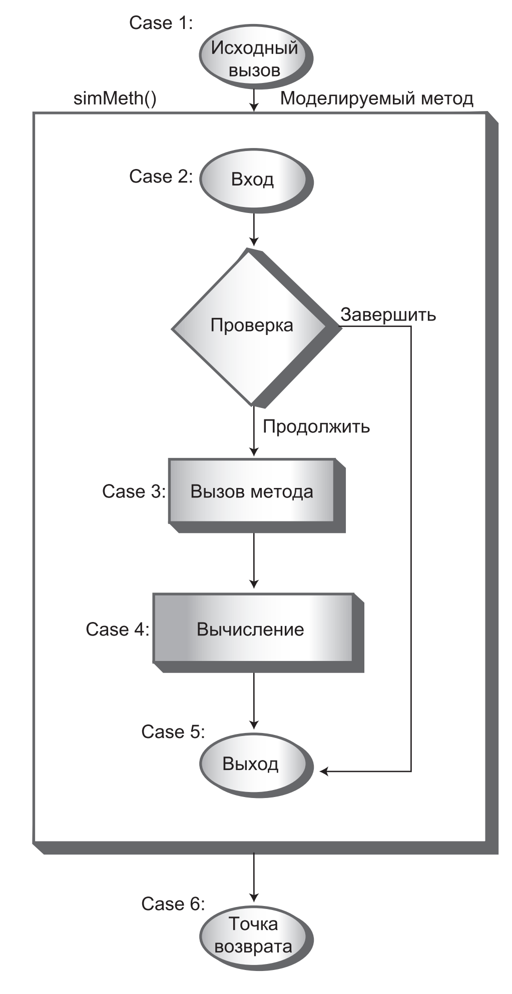


Рис. 6.19. Секции case и метод step

Префикс case показывает, какая секция кода выполняется в настоящий момент. Также выводится содержимое стека (состоящее из объектов Params со значением n и адресом возврата). Вход в метод simMeth() выполняется четыре раза (case 2); столько же раз выполняется выход (case 5). Только с первым возвратом в theAnswer начинают накапливаться результаты вычислений.

Что дальше?

В программе stackTriangle.java (листинг 6.7) представлен более или менее систематический способ преобразования программы, использующей рекурсию, в программу, использующую стек. Подобное преобразование возможно в любой программе, использующей рекурсию.

Немного поработав над кодом, можно усовершенствовать его, упростить и даже убрать конструкцию switch для повышения эффективности. Однако на практике обычно бывает разумнее переработать алгоритм с самого начала, используя стековую реализацию вместо рекурсивной. В листинге 6.8 представлен результат такой переработки для метода triangle().

Листинг 6.8. Программа stackTriangle2.java

```
// stackTriangle2.java
// Вычисление треугольных чисел (стек вместо рекурсии)
// Запуск программы: C>java StackTriangle2App
import java.io.*;                 // Для ввода/вывода
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;        // Размер массива
   private int[] stackArray;
   private int top;            // Вершина стека
//--------------------------------------------------------------
   public StackX(int s)         // Конструктор
      {
      maxSize = s;
      stackArray = new int[maxSize];
      top = -1;
      }
//--------------------------------------------------------------
   public void push(int p)     // Размещение элемента на вершине стека
      { stackArray[++top] = p; }
//--------------------------------------------------------------
   public int pop()            // Извлечение элемента с вершины стека
      { return stackArray[top--]; }
//--------------------------------------------------------------
   public int peek()           // Чтение элемента с вершины стека
      { return stackArray[top]; }
//--------------------------------------------------------------
```

продолжение ⇨

Листинг 6.6 (*продолжение*)

```
   public boolean isEmpty()    // true, если стек пуст
      { return (top == -1); }
//--------------------------------------------------------------
   }  // Конец класса StackX
////////////////////////////////////////////////////////////////
class StackTriangle2App
   {
   static int theNumber;
   static int theAnswer;
   static StackX theStack;

   public static void main(String[] args) throws IOException
      {
      System.out.print("Enter a number: ");
      theNumber = getInt();
      stackTriangle();
      System.out.println("Triangle="+theAnswer);
      }
//-------------------------------------------------------------
   public static void stackTriangle()
      {
      theStack = new StackX(10000);    // Создание стека

      theAnswer = 0;                   // Инициализация ответа

      while(theNumber > 0)             // Пока счетчик не уменьшится до 1
         {
         theStack.push(theNumber);     // Занесение элемента в стек
         --theNumber;                  // Уменьшение счетчика
         }
      while( !theStack.isEmpty() )     // Пока в стеке остаются элементы
         {
         int newN = theStack.pop();    // Извлечение значения
         theAnswer += newN;            // Суммирование с ответом
         }
      }
//-------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//-------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
```

```
      return Integer.parseInt(s);
      }
//------------------------------------------------------------
   }  // Конец класса StackTriangle2App
```

Два коротких цикла while в методе stackTriangle() заменяют весь метод step() в программе stackTriangle.java. Конечно, в этой простой программе стек можно полностью исключить, заменив его простым циклом. В более сложных алгоритмах без стека не обойтись.

Чтобы понять, какое из решений (рекурсивный метод, стековая реализация, простой цикл) наиболее эффективно в конкретной ситуации, нередко приходится экспериментировать.

Интересные применения рекурсии

В этом разделе рассмотрены примеры задач, для решения которых может применяться рекурсия. Разнообразие примеров наглядно показывает, что рекурсия порой встречается в самых неожиданных местах. Мы рассмотрим три проблемы: возведение числа в степень, укладка вещей в рюкзак и набор команды альпинистов. В тексте объясняются основные концепции, а реализация остается читателю для самостоятельной работы.

Возведение числа в степень

Многие карманные калькуляторы позволяют возвести число в произвольную степень. Обычно на них имеется клавиша x^y; знак «^» указывает, что число *x* возводится в степень *y*. Как бы вы выполняли эти вычисления, если бы на калькуляторе не было такой клавиши? Конечно, можно умножить *x* на себя *y* раз. Другими словами, если значение *x* равно 2, а *y* — 8 (2^8), следует выполнить операцию $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2$. Однако при больших значениях *y* решение получается громоздким. Нет ли более быстрого способа?

Одно из возможных решений основано на группировке множителей, чтобы вместо 2 по возможности использовались числа, кратные 2. Для примера возьмем 2^8. В конечном итоге число 2 необходимо умножить само на себя 8 раз. Начнем с $2 \times 2 = 4$. Две «двойки» перемножены, остается еще шесть. Однако теперь у нас появилось новое число для работы: 4. Умножаем, получаем $4 \times 4 = 16$. Четыре «двойки» перемножены (потому что каждое число 4 — это 2×2). Необходимо использовать еще четыре раза по 2, но на предыдущем шаге было получено число 16, а операция $16 \times 16 = 256$ задействует в точности восемь «двоек» (потому что каждый множитель 16 состоит из четырех «двоек»).

Итак, нам удалось вычислить значение 2^8 всего с тремя операциями умножения вместо семи. Сложность операции сократилась с $O(N)$ до $O(\log N)$.

Можно ли сформулировать этот процесс в виде алгоритма, понятного для компьютера? Схема основана на математической формуле $x^y = (x^2)^{y/2}$. В нашем

примере $2^8 = (2^2)^{8/2}$ или $2^8 = (2^2)^4$ — как известно, возведение степени в другую степень эквивалентно перемножению показателей степеней.

Но если наш компьютер не умеет возводить в степень, вычислить результат $(2^2)^4$ не удастся. Нельзя ли преобразовать это выражение в другое, в котором используется только умножение? Давайте заменим 2^2 новой переменной.

Допустим, $2^2 = a$. В этом случае выражение 2^8 эквивалентно $(2^2)^4$, то есть a^4. Но в соответствии с исходной формулой a^4 записывается в виде $(a^2)^2$, так что $2^8 = (a^2)^2$.

И снова $а^2$ заменяется новой переменной (допустим, $a^2 = c$); выражение $(c)^2$ можно записать в виде $(c^2)^1$, причем результат этого выражения по-прежнему равен 2^8.

В итоге мы пришли к задаче, которая решается простым умножением c на c.

Описанная схема легко воплощается в виде рекурсивного метода (назовем его `power()`) для вычисления степеней. Метод получает x и y в аргументах и возвращает x^y. О вспомогательных переменных можно не беспокоиться, потому что x и у присваиваются новые значения при каждом рекурсивном вызове. В аргументах передаются значения $x \times x$ и $y/2$. Например, для $x = 2$ и $y = 8$ последовательность аргументов и возвращаемых значений будет выглядеть так:

```
x=2, y=8
x=4, y=4
x=16, y=2
x=256, y=1
Returning 256, x=256, y=1
Returning 256, x=16, y=2
Returning 256, x=4, y=4
Returning 256, x=2, y=8
```

Когда значение y равно 1, происходит возврат из рекурсивного метода. Ответ 256 передается обратно по цепочке вызовов без изменений.

В рассмотренном примере y остается четным числом на протяжении всей серии разбиений. На практике это часто бывает не так. Модификация алгоритма для нечетных y выглядит так: в рекурсии используется только целочисленное деление, а при делении y на 2 остаток игнорируется. Однако в процессе возврата каждый раз, когда y является нечетным числом, выполняется дополнительное умножение на x. Последовательность вызовов для вычисления 3^{18} выглядит следующим образом:

```
x=3, y=18
x=9, y=9
x=81, y=4
x=6561, y=2
x=43046721, y=1
Returning 43046721, x=43046721, y=1
Returning 43046721, x=6561, y=2
Returning 43046721, x=81, y=4
Returning 387420489, x=9, y=9  // Значение y нечетно, умножаем на x
Returning 387420489, x=3, y=18
```

Задача о рюкзаке

Задача о рюкзаке[1] относится к классическим задачам программирования. В простейшей форме этой задачи рюкзак требуется заполнить предметами различного веса для достижения заданного суммарного веса. Укладывать в рюкзак все предметы не обязательно.

Допустим, собранный рюкзак должен весить ровно 20 кг, и у вас имеется пять предметов с весом 11, 8, 7, 6 и 5 кг. При малом количестве предметов человек неплохо справляется с этой задачей простым перебором. Вероятно, вы уже увидели, что только комбинация 8, 7 и 5 дает в сумме 20. Чтобы эту задачу мог решить компьютер, ему необходимо предоставить более подробные инструкции. Алгоритм решения выглядит так:

1. Если в какой-то момент суммарный вес всех выбранных предметов достигнет целевого, работа завершена.
2. Выбрать первый предмет. Общий вес остальных предметов должен быть равен разности между целевым весом рюкзака и весом первого предмета; эта разность становится новым целевым весом.
3. Последовательно перебрать все комбинации остальных предметов. Впрочем, реально обратите внимание на то, что перебирать все без исключения комбинации не обязательно — если суммарный вес предметов превышает целевой, добавление новых предметов можно прекратить.
4. Если ни одна из комбинаций не сработала, отложить первый предмет и повторить весь процесс для второго.
5. Продолжить с третьим предметом и т. д., пока не будут проверены все комбинации. После завершения перебора можно быть уверенным в том, что решения не существует.

В приведенном выше примере начнем с 11. Суммарный вес остальных предметов должен быть равен 9 (20 – 11). Начинаем с 8. На долю остальных предметов остается 1 (9 – 8). Начинаем с 7; этот вес больше 1. Проверяем 6, затем 5 — все еще слишком много. Перебор завершен, и мы знаем, что ни одна комбинация с 8 в сумме не даст 9. Проверяем 7 и ищем комбинацию с целевым весом 2 (9 – 7). Перебор продолжается аналогичным образом:

```
Items: 11, 8, 7, 6, 5
==========================================
11          // Цель = 20, 11 мало
11, 8       // Цель = 9, 8 мало
11, 8, 7    // Цель = 1, 7 много
11, 8, 6    // Цель = 1, 6 много
11, 8, 5    // Цель = 1, 5 много. Предметов больше нет
11, 7       // Цель = 9, 7 мало
11, 7, 6    // Цель = 2, 6 много
11, 7, 5    // Цель = 2, 5 много. Предметов больше нет
11, 6       // Цель = 9, 6 мало
```

[1] Также называемая «задачей о ранце». — *Примеч. перев.*

```
11, 6, 5   // Цель = 3, 5 много. Предметов больше нет
11, 5      // Цель = 9, 5 мало. Предметов больше нет
8,         // Цель = 20, 8 мало
8, 7       // Цель = 12, 7 мало
8, 7, 6    // Цель = 5, 6 много
8, 7, 5    // Цель = 5, 5 точное совпадение. Успех!
```

Рекурсивный метод берет первый предмет, и если его вес меньше целевого, снова вызывает себя с новым целевым весом, чтобы проверить суммы всех оставшихся предметов.

Комбинации и выбор команды

В математике *комбинацией* называется совокупность элементов, которые могут следовать в произвольном порядке. Допустим, имеется группа из пяти альпинистов A, B, C, D и E. Из этой группы необходимо выбрать команду из трех человек для покорения крутых и заледенелых склонов пика Анаконда. Однако вас беспокоит психологическая совместимость участников команды, поэтому вы решаете составить список всех возможных команд, то есть сгенерировать все возможные комбинации из трех альпинистов. Было бы удобно, если бы компьютер распечатал список комбинаций для вас. Программа должна вывести 10 возможных комбинаций:

```
ABC, ABD, ABE, ACD, ACE, ADE, BCD, BCE, BDE, CDE
```

Как написать такую программу? Оказывается, у задачи имеется элегантное рекурсивное решение. Все комбинации делятся на две группы: начинающиеся с A и начинающиеся с других букв. Для удобства концепция выделения 3-х участников в группе из 5 человек будет обозначаться записью вида (5, 3). Допустим, размер группы равен n, а размер команды равен k. Согласно теореме из области комбинаторики

$$(n, k) = (n - 1, k - 1) + (n - 1, k).$$

В нашем примере с отбором 3-х участников в группе из 5 получается:

$$(5, 3) = (4, 2) + (4, 3).$$

Нам удалось разбить задачу на две меньших. Вместо того чтобы выбирать в группе размера 5, мы выбираем дважды в группе размера 4: сначала все возможные варианты выбора 2-х участников в группе из 4, а затем все варианты выбора 3-х участников в группе из 4.

Всего существует 6 возможных вариантов выбора 2-х участников в группе из 4. В записи (4, 2) (для удобства назовем ее *левым слагаемым*) это комбинации

BC, BD, BE, CD, CE, DE

В этих комбинациях отсутствует участник A; чтобы команда состояла из 3-х человек, к комбинациям присоединяется префикс A:

ABC, ABD, ABE, ACD, ACE, ADE

Отобрать 3-х участников в группе из 4-х можно четырьмя способами. Набор (4, 3) (*правое слагаемое*) состоит из комбинаций

BCD, BCE, BDE, CDE

Прибавляя 4 комбинации правого слагаемого к 6 комбинациям левого слагаемого, мы получаем 10 комбинаций для (5, 3).

Аналогичный процесс декомпозиции может быть применен к каждой из групп размера 4. Например, результат (4, 2) равен сумме (3, 1) + (3, 2). Как видите, ситуация естественным образом подходит для применения рекурсии.

Задачу можно представить в виде дерева с вершиной (5, 3), узлами следующего уровня (4, 3) и (4, 2) и т. д.; каждый узел дерева соответствует рекурсивному вызову. На рис. 6.20 показано, как выглядит дерево для примера (5, 3).

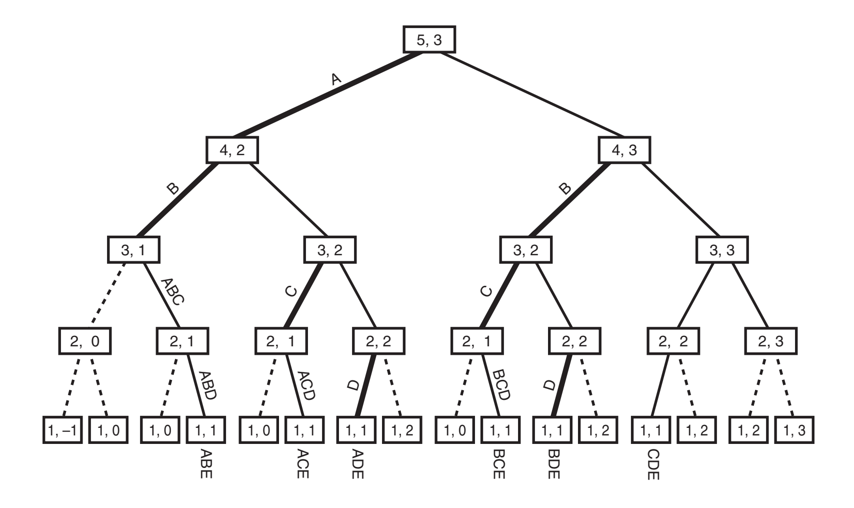


Рис. 6.20. Выбор команды размера 3 в группе из 5 участников

Базовыми ограничениями являются комбинации, не имеющие смысла: те, у которых один из компонентов равен 0, и те, у которых размер команды больше размера группы. Комбинация (1, 1) допустима, но попытки ее дальнейшего деления смысла не имеют. На рисунке базовые ограничения выделены пунктирными линиями; вместо дальнейшего раскрытия эти вызовы просто возвращают управление.

Глубина рекурсии соответствует участникам группы: узел верхнего уровня представляет участника A, два узла следующего уровня — участника B и т. д. Если группа состоит из 5 участников, дерево будет содержать 5 уровней.

В процессе перемещения вниз по дереву необходимо запоминать последовательность перебираемых участников. При вызове для левого слагаемого узел, который вы покидаете, отмечается включением соответствующей буквы в последовательность. Такие вызовы и буквы, включаемые в последовательность, отмечены на рисунке жирными линиями.

При рекурсивных вызовах для левых слагаемых программа ничего не выводит, а при вызовах для правых слагаемых необходимо проверить последовательность; если вы находитесь в допустимом узле дерева и добавление одного участника завершит состав команды, включите этот узел в последовательность и выведите полный набор.

Итоги

- Рекурсивный метод повторно вызывает сам себя с разными аргументами.
- При некоторых значениях аргументов рекурсивный метод возвращает управление без дальнейших рекурсивных вызовов. Это называется *базовым ограничением*.
- При возврате управления рекурсивным вызовом наибольшей глубины происходит процесс «раскрутки» — незавершенные вызовы завершаются по убыванию вложенности, от внутренних вызовов к исходному.
- Треугольным числом называется сумма самого числа со всеми меньшими числами (естественно, речь идет о *натуральных* числах). Например, треугольное число с номером 4 равно 10, потому что 4 + 3 + 2 + 1 = 10.
- Факториалом числа называется произведение самого числа и всех меньших чисел. Например, факториал 4 равен $4 \times 3 \times 2 \times 1 = 24$.
- И треугольные числа, и факториалы могут вычисляться как рекурсивным методом, так и в простом цикле.
- Анаграммы слова (все возможные комбинации n букв слова) могут строиться рекурсивно, посредством циклического сдвига и построением анаграмм правых $n - 1$ букв.
- Двоичный поиск также может выполняться рекурсивно; программа проверяет, в какой половине отсортированного диапазона находится ключ, а затем выполняет операцию деления и проверки с соответствующей половиной.
- Головоломка «Ханойская башня» состоит из трех стержней и произвольного количества дисков.
- В рекурсивном решении головоломки «Ханойская башня» все диски поддерева, кроме самого нижнего, перемещаются на промежуточный стержень, затем нижний диск перемещается на приемный стержень, и наконец, поддерево перемещается на приемный стержень.
- В результате слияния двух отсортированных массивов создается третий массив, содержащий все элементы обоих массивов в порядке сортировки.
- При сортировке слиянием 1-элементные подмассивы большего массива объединяются в 2-элементные подмассивы, которые в свою очередь объединяются в 4-элементные подмассивы и т. д., пока не будет отсортирован весь массив.
- Сортировка слиянием выполняется за время $O(N \times \log N)$.

- Для выполнения сортировки слиянием требуется рабочая область, размер которой совпадает с размером исходного массива.
- Для вычисления треугольных чисел, факториалов, построения анаграмм и двоичного поиска рекурсивный метод содержит только один вызов самого себя. (В коде двоичного поиска таких вызовов два, но при каждом конкретном выполнении кода метода используется только один из них.)
- В «Ханойской башне» и сортировке слиянием рекурсивный метод содержит два вызова самого себя.
- Любая операция, которая может выполняться посредством рекурсии, также может быть реализована на базе стека.
- Рекурсивное решение может оказаться неэффективным. Иногда его удается заменить простым циклом или решением на базе стека.

Вопросы

Следующие вопросы помогут читателю проверить качество усвоения материала. Ответы на них приведены в приложении В.

1. Предположим, пользователь вводит 10 в программе `triangle.java` (листинг 6.1). Какое максимальное количество «копий» метода `triangle()` (фактически копий его аргумента) может существовать в любой отдельный момент времени?
2. Где хранятся копии аргумента, упоминаемые в вопросе 1?
   a) В переменной метода `triangle()`.
   b) В поле класса `TriangleApp`.
   c) В переменной метода `getString()`.
   d) В стеке.
3. Предположим, пользователь вводит 10 в вопросе 1. Чему будет равно значение *n*, когда метод `triangle()` впервые вернет значение, отличное от 1?
4. Ситуация та же, что в вопросе 1. Чему будет равно значение *n*, когда метод `triangle()` будет готов вернуть управление методу `main()`?
5. В методе `triangle()` возвращаемые значения хранятся в стеке (Да/Нет).
6. В программе `anagram.java` (листинг 6.2) на некоторой глубине рекурсии версия метода `doAnagram()` работает со строкой «led». С какими буквами будет работать новая версия, которую вызовет этот метод?
7. Примеры замены цикла рекурсией были представлены в программе `orderedArray.java` (листинг 2.4) и рекурсивной версии `binarySearch.java` (листинг 6.3). Какое из следующих утверждений *ложно*?
   a) Обе программы последовательно делят диапазон надвое.
   b) Если ключ не найден, версия с циклом возвращает управление при пересечении границ диапазонов, а рекурсивная версия — при достижении нижнего уровня рекурсии.

c) Если ключ найден, версия с циклом возвращает управление из всего метода, а рекурсивная версия — только с одного уровня рекурсии.

d) В рекурсивной версии диапазон для поиска должен задаваться в аргументах, а в версии с циклом это не обязательно.

8. Что именно заменяет цикл из некурсивной версии программы в методе `recFind()` программы `binarySearch.java` (листинг 6.3)?

   a) Метод `recFind()`.

   b) Аргументы `recFind()`.

   c) Рекурсивные вызовы `recFind()`.

   d) Вызов `recFind()` из `main()`.

9. Программа `binarySearch.java` — пример решения задачи методом ___________.

10. Что уменьшается при повторных рекурсивных вызовах метода `recFind()`?

11. Что уменьшается при повторных рекурсивных вызовах в программе `towers.java` (листинг 6.4)?

12. В алгоритме программы `towers.java` используется:

    a) «деревья» для хранения данных;

    b) незаметное подкладывание меньших дисков под большие;

    c) смена исходных и приемных стержней;

    d) перемещение одного меньшего диска с последующим перемещением стопки больших дисков.

13. Какое из следующих утверждений *ложно* по отношению к методу `merge()` программы `merge.java` (листинг 6.5)?

    a) Его алгоритм способен поддерживать массивы разных размеров.

    b) Для определения места для размещения следующего элемента он должен провести поиск в приемном массиве.

    c) Он не рекурсивен.

    d) Он в цикле отбирает наименьший элемент независимо от того, в каком из массивов он находится.

14. Недостаток сортировки слиянием:

    a) нерекурсивная природа;

    b) более высокие затраты памяти;

    c) хотя сортировка слиянием по скорости превосходит сортировку методом вставок, она значительно медленнее быстрой сортировки;

    d) сложность реализации.

15. Кроме циклов, для замены рекурсии также часто используется _____________.

Упражнения

Упражнения помогут вам глубже усвоить материал этой главы. Программирования они не требуют.

1. В программе `triangle.java` (листинг 6.1) удалите код базового ограничения (`if(n==1)`, `return 1;` и `else`). Запустите программу и проверьте, что произойдет.
2. Используя приложение Towers Workshop в ручном режиме, решите головоломку для семи и более дисков.
3. Перепишите метод `main()` программы `mergeSort.java` (листинг 6.6), чтобы массив заполнялся сотнями тысяч случайных чисел. Запустите программу для сортировки этих чисел и сравните ее скорость с другими алгоритмами сортировки из главы 3, «Простая сортировка».

Программные проекты

В этом разделе читателю предлагаются программные проекты, которые укрепляют понимание материала и демонстрируют практическое применение концепций этой главы.

6.1. Предположим, вы купили дешевый карманный компьютер, который не умеет выполнять операцию умножения — только сложение. Чтобы выйти из положения, вы пишете рекурсивный метод `mult()`, который выполняет умножение x на y посредством сложения x с самим собой y раз. Метод получает x и y в аргументах, и возвращает их произведение. Напишите такой метод и программу `main()`, в которой он вызывается. Когда выполняется сложение — когда метод вызывает сам себя или когда он возвращает управление?

6.2. В главе 8, «Двоичные деревья», рассматриваются двоичные деревья, у которых каждая ветвь имеет (теоретически) ровно две подветви. Если попытаться изобразить двоичное дерево на экране в символьном режиме, в первой строке будет один узел, во второй — два, затем 4, 8, 16 и т. д. Например, при длине в 16 символов вывод будет выглядеть так:

```
--------X-------
----X-------X---
--X---X---X---X-
-X-X-X-X-X-X-X-X
XXXXXXXXXXXXXXXX
```

(Конечно, нижнюю строку следовало бы сдвинуть на пол-символа вправо, но в символьном режиме вывода это невозможно.) Для вывода дерева можно воспользоваться рекурсивным методом `makeBranches()` с аргументами `left` и `right`, определяющими конечные точки горизонтального диапазона. При первом входе в метод `left` содержит 0, а `right` — количество символов (включая дефисы) в каждой строке минус один. Метод выводит X в центре диапазона, после чего вызывает себя дважды: для левой и для правой половины диапазона. Когда диапазон умень-

шается ниже некоторого предела, метод возвращает управление. Все дефисы и X сохраняются в массиве, содержимое которого выводится за один раз — скажем, при вызове метода `display()`. Напишите метод `main()`, который выводит дерево, вызывая `makeBranches()` и `display()`. Метод `main()` должен сам определять длину строки вывода в символах (32, 64 и т. д.) Проследите за тем, чтобы массив для хранения выводимых символов был не больше необходимого. Каким образом количество строк (5 на приведенном рисунке) связано с длиной строки?

6.3. Реализуйте рекурсивное возведение числа в степень (см. раздел «Возведение числа в степень» этой главы). Напишите рекурсивную функцию `power()` и метод `main()` для ее тестирования.

6.4. Напишите программу, которая решает задачу о рюкзаке для произвольной емкости рюкзака и набора предметов. Веса предметов должны храниться в массиве. Подсказка: в аргументах рекурсивной функции `knapsack()` передаются целевой вес и индекс массива, с которого начинаются оставшиеся предметы.

6.5. Реализуйте рекурсивное решение задачи вывода всех комбинаций (n объектов из k возможных). Напишите рекурсивный метод `showTeams()` и метод `main()`, который запрашивает у пользователя размеры группы и команды и передает их в аргументах методу `showTeam()`, выводящему все возможные комбинации.

Глава 7

Нетривиальная сортировка

Простые методы сортировки, представленные в главе 3, — пузырьковая, сортировка методом выбора и вставок — легко реализуются, но работают относительно медленно. В главе 6, «Рекурсия», была описана сортировка слиянием. По скорости она заметно превосходит простые алгоритмы сортировки, но занимает вдвое больше памяти по сравнению с исходным массивом; эта ее особенность нередко оказывается серьезным недостатком.

В этой главе рассматриваются два нетривиальных алгоритма: сортировка Шелла и быстрая сортировка. Оба работают значительно быстрее простых алгоритмов: сортировка Шелла выполняется за время $O(N \times (\log N)^2)$, а быстрая сортировка — за время $O(N \times \log N)$. В отличие от сортировки слиянием, ни один из этих алгоритмов не требует значительных затрат памяти. Сортировка Шелла реализуется почти так же просто, как сортировка слиянием, а быстрая сортировка является самой эффективной из всех алгоритмов сортировки общего назначения. Глава завершается кратким описанием поразрядной сортировки — необычного и интересного подхода к решению задачи сортировки.

Начнем с сортировки Шелла. Алгоритм быстрой сортировки основан на концепции разбиения, поэтому разбиение будет рассмотрено отдельно, еще до рассмотрения собственно быстрой сортировки.

Сортировка Шелла

Алгоритм назван в честь Дональда Л. Шелла — специалиста в области компьютерных технологий, который опубликовал описание этого способа сортировки в 1959 году. Алгоритм Шелла основан на сортировке методом вставок, но он обладает новой особенностью, кардинально улучшающей скорость сортировки.

Сортировка Шелла хорошо подходит для массивов среднего размера — например, до нескольких тысяч элементов (в зависимости от конкретной реализации). По скорости она уступает быстрой сортировке и другим алгоритмам, выполняемым за время $O(N \times \log N)$, поэтому для очень больших объемов данных она не оптимальна. Тем не менее сортировка Шелла заметно быстрее алгоритмов $O(N^2)$ — таких, как сортировки методом выбора и вставок. К тому же она отличается простотой реализации: код получается простым и компактным.

Быстродействие в худшем случае незначительно уступает среднему быстродействию. (Позднее в этой главе мы увидим, что для быстрой сортировки быстродействие в худшем случае может быть значительно хуже среднего, если не принять необходимые меры предосторожности.) Некоторые эксперты (ссылка на книгу

Седжвика приведена в приложении Б) рекомендуют почти в любой ситуации начинать с сортировки Шелла, а затем переключаться на более сложный алгоритм (такой, как быстрая сортировка) только в том случае, если метод Шелла на практике окажется слишком медленным.

Сортировка методом вставок: слишком много копирования

Так как алгоритм Шелла основан на сортировке методом вставок, возможно, вам стоит вернуться к разделу «Сортировка методом вставки» главы 3. Напомним, что в ходе выполнения вставки элементы слева от маркера обладают внутренней упорядоченностью (то есть отсортированы в пределах своего набора), а элементы справа не упорядочены. Алгоритм извлекает элемент, на который указывает маркер, и сохраняет его во временной переменной. Затем, начиная с элемента слева от освободившейся ячейки, он сдвигает отсортированные элементы вправо, пока не появится возможность вставки содержимого временной переменной с сохранением порядка сортировки.

И в этом процессе кроется основной недостаток сортировки методом вставок. Предположим, у правого края массива — там, где должны быть большие элементы — находится малый элемент. Чтобы переместить его на подходящее место где-то слева, необходимо сдвинуть все промежуточные элементы (то есть элементы между его фактическим и итоговым местом) на одну позицию вправо. Это потребует выполнения где-то до N операций копирования всего для одного элемента. Не все элементы придется перемещать на полные N позиций, но в среднем один элемент будет перемещаться на $N/2$ позиций. Итого $N/2$ сдвигов будут выполняться N раз, а среднее количество операций копирования составит $N^2/2$. Таким образом, сортировка методом вставок обладает эффективностью $O(N^2)$.

Быстродействие можно улучшить, если бы меньший элемент можно было сдвинуть к левому краю массива без сдвига всех промежуточных элементов.

N-сортировка

Сортировка Шелла выполняет «дальние» сдвиги, сортируя разобщенные элементы посредством сортировки методом вставок. После сортировки этих элементов сортируются элементы, находящиеся на меньшем расстоянии и т. д. Расстояние между элементами при такой сортировке называется *приращением* и традиционно обозначается буквой h. На рис. 7.1 показан первый шаг процесса сортировки 10-элементного массива с приращением 4. Здесь сортируются элементы 0, 4 и 8. После того как будут отсортированы элементы 0, 4 и 8, алгоритм сдвигается на одну ячейку и сортирует элементы 1, 5 и 9. Процесс продолжается до тех пор, пока все элементы не пройдут «4-сортировку», то есть все элементы, находящиеся на расстоянии 4-х ячеек, не будут отсортированы между собой. Этот процесс изображен на рис. 7.2 (в более компактном визуальном представлении).

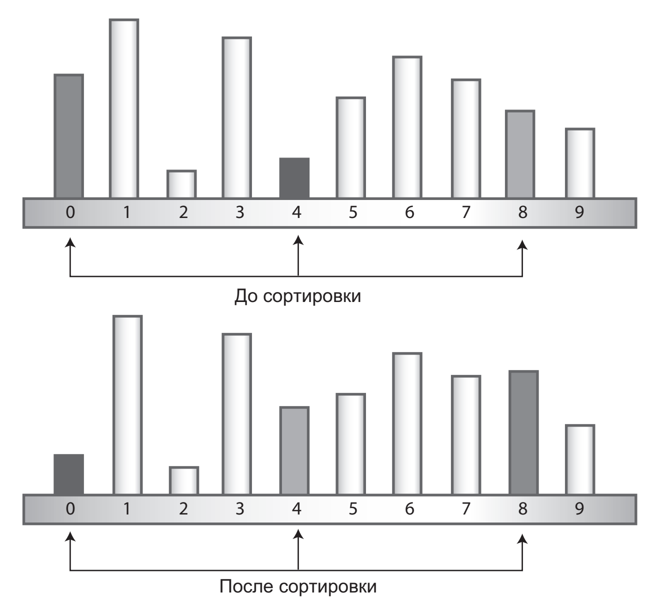


Рис. 7.1. 4-сортировка элементов 0, 4 и 8

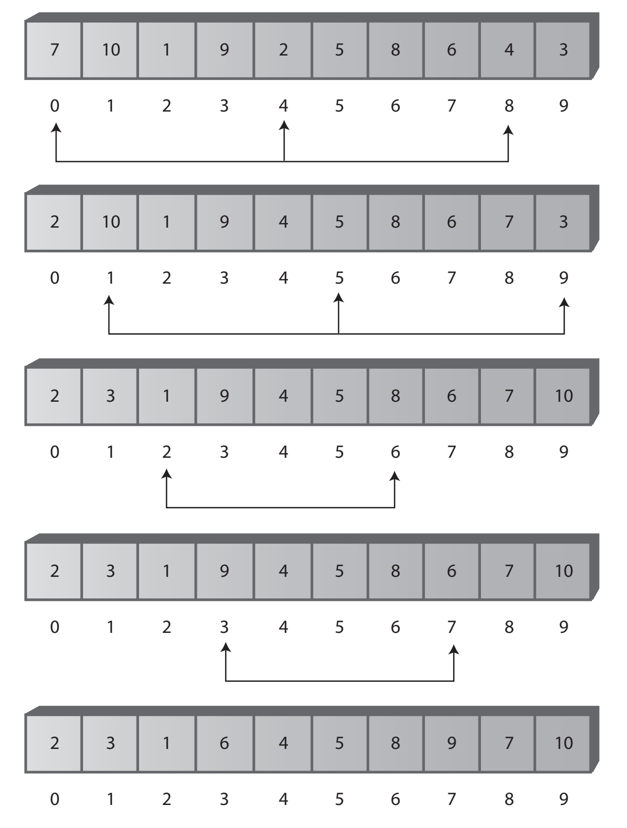


Рис. 7.2. Полная 4-сортировка

После полной 4-сортировки массив можно рассматривать как совокупность четырех подмассивов: (0,4,8), (1,5,9), (2,6) и (3,7), каждый из которых полностью отсортирован. Элементы подмассивов чередуются, но в остальном подмассивы полностью независимы друг от друга.

Обратите внимание: в этом конкретном примере в конце 4-сортировки ни один элемент не находится на расстоянии более двух ячеек от той позиции, где он находился бы при полной сортировке массива. Именно эта особенность — «почти» отсортированный массив — и является секретом сортировки Шелла. Создание чередующихся наборов элементов с внутренней сортировкой сводит к минимуму объем работы, которую необходимо проделать для завершения сортировки.

Теперь, как было замечено в главе 3, сортировка методом вставок чрезвычайно эффективна для почти отсортированных массивов. Если для завершения сортировки достаточно сместить каждый элемент на одну-две ячейки, сортировка выполняется почти за время $O(N)$. Таким образом, 4-отсортированный массив можно 1-отсортировать посредством обычной сортировки методом вставок. Комбинация 4-сортировки с 1-сортировкой работает намного быстрее, чем простое применение сортировки методом вставок без предварительной 4-сортировки.

Сокращение интервалов

Ранее была продемонстрирована сортировка массива из 10 элементов с исходным интервалом в 4 ячейки. Для массивов большего размера предварительная сортировка должна начинаться с намного большего интервала, который затем последовательно сокращается вплоть до 1.

Например, в массиве из 1000 элементов может быть проведена 364-сортировка, затем 121-сортировка, 40-сортировка, 13-сортировка, 4-сортировка и, наконец, 1-сортировка. Серия чисел, используемых при генерировании интервалов (в данном примере 364, 121, 40, 13, 4, 1), называется *интервальной последовательностью*. Приведенная интервальная последовательность, приписываемая Кнуту (см. приложение Б), весьма популярна. В обратной форме, начиная с 1, она генерируется по рекурсивной формуле

$$h = 3 \times h + 1.$$

(начальное значение h равно 1). Процесс генерирования последовательности по формуле представлен в первых двух столбцах табл. 7.1.

Также существуют другие методы построения интервальных последовательностей; мы вернемся к этой теме чуть позднее. А сначала давайте исследуем, как работает сортировка Шелла с использованием последовательности Кнута.

В алгоритме сортировки сначала в коротком цикле вычисляется исходный интервал. В качестве первого значения h используется 1, а затем по формуле $h = h \times 3 + 1$ генерируется последовательность 1, 4, 13, 40, 121, 364 и т. д. Процесс завершается тогда, когда величина интервала превышает размер массива. Для массива из 1000 элементов седьмое число в последовательности (1093) оказывается слишком большим; соответственно процесс сортировки начинается с шестого

числа (364). Затем при каждой итерации внешнего цикла метода сортировки интервал сокращается по формуле, обратной по отношению к приведенной:

$$h = (h{-}1)/3.$$

Полученные числа приведены в третьем столбце табл. 7.1. По обратной формуле генерируется обратная последовательность 364, 121, 40, 13, 4, 1. Каждое из этих чисел, начиная с 364, используется для *n*-сортировки массива. Выполнение 1-сортировки завершает работу алгоритма.

Таблица 7.1. Интервальная последовательность Кнута

h	3*h+1	(h – 1)/3
1	4	
4	13	1
13	40	4
40	121	13
121	364	40
364	1093	121
1093	3280	364

Приложение Shellsort Workshop

Приложение Shellsort Workshop показывает, как работает этот алгоритм сортировки. На рис. 7.3 показано, как выглядит окно приложения после выполнения 4-сортировки, непосредственно перед началом 1-сортировки.

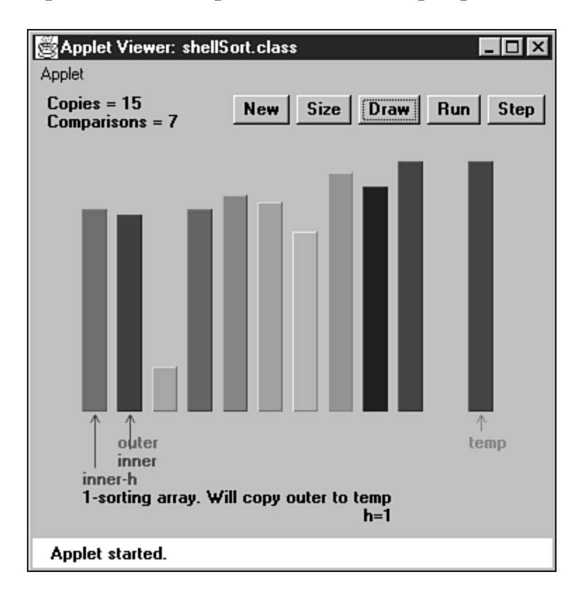


Рис. 7.3. Приложение Shellsort Workshop

При пошаговом выполнении алгоритма вы заметите, что приведенные ранее объяснения были слегка упрощены. В процессе 4-сортировки не используются упоминавшиеся последовательности (0,4,8), (1,5,9), (2,6) и (3,7). Вместо этого сначала сортируются первые два элемента каждой группы из трех, затем первые два элемента второй группы и т. д. После того как первые два элемента во всех группах будут отсортированы, алгоритм возвращается и сортирует группы из трех элементов. Таким образом, фактически используемая последовательность сортировки имеет вид (0,4), (1,5), (2,6), (3,7), (0,4,8), (1,5,9).

На первый взгляд кажется более логичным выполнить 4-сортировку каждого полного подмассива — (0,4), (0,4,8), (1,5), (1,5,9), (2,6), (3,7), — однако первая схема обеспечивает более эффективную работу с индексами.

Для 10 элементов сортировка Шелла не особенно эффективна; по количеству перестановок и сравнений она мало чем отличается от сортировки методом вставок. Впрочем, уже при 100 элементах выигрыш становится существенным.

Поучительно запустить приложение для 100 элементов, отсортированных в обратном порядке. (Как и в главе 3, первое нажатие кнопки `New` создает случайную последовательность элементов, а второе — последовательность, отсортированную в обратном порядке.) На рис. 7.4 показано, как выглядят элементы после первого прохода, то есть полной 40-сортировки массива. На рис. 7.5 представлена ситуация после следующего прохода, когда массив прошел 13-сортировку. С каждым новым значением *h* массив приближается к состоянию сортировки.

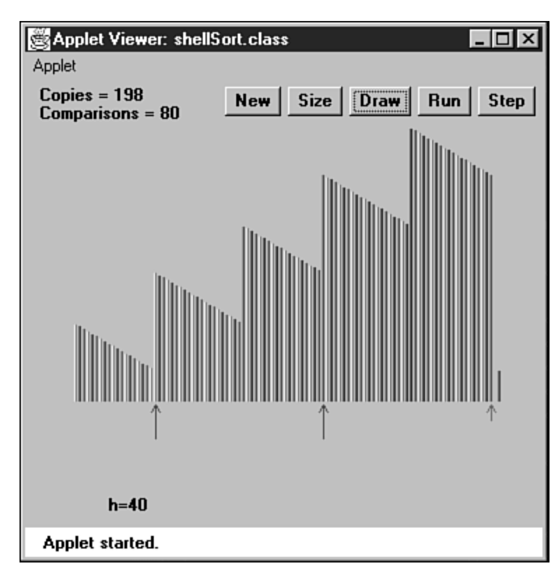


Рис. 7.4. После 40-сортировки

Почему сортировка Шелла работает быстрее сортировки методом вставки, на которой она базируется? При больших значениях *h* количество элементов на один проход невелико, а элементы перемещаются на большие расстояния. С уменьшением *h* количество элементов на каждую итерацию возрастает, но и элементы

находятся ближе к своим конечным позициям в порядке сортировки, поэтому алгоритм по эффективности превосходит сортировку методом вставки. Сочетание этих особенностей делает сортировку такой эффективной.

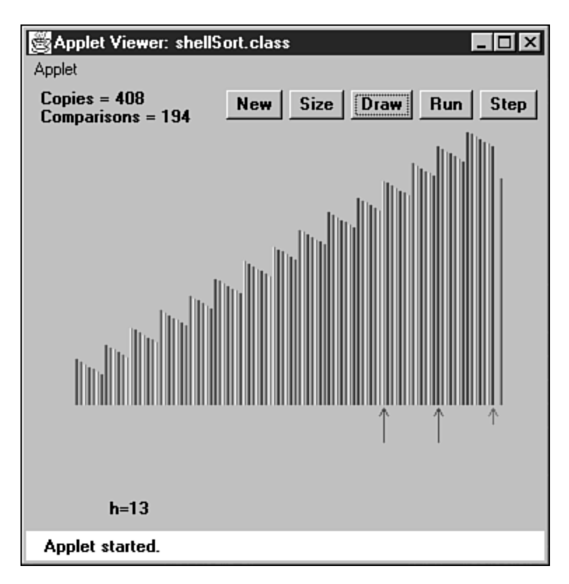


Рис. 7.5. После 13-сортировки

Обратите внимание на то, что последующие сортировки (с меньшими значениями h) не отменяют работы предыдущих сортировок (с большими значениями h). Например, массив, который прошел 40-сортировку, остается 40-отсортированным и после 13-сортировки. Если бы это свойство не выполнялось, то алгоритм Шелла был бы неработоспособным.

Реализация сортировки Шелла на языке Java

Реализация сортировки Шелла на Java ненамного сложнее реализации сортировки методом вставки. Фактически в коде сортировки методом вставки 1 в соответствующих местах заменяется на h, а также добавляется формула построения интервальной последовательности. В нашей реализации сортировка Шелла оформлена в виде метода `shellSort()` класса `ArraySh` — версии класса массива из главы 2, «Массивы». Полный код программы `shellSort.java` приведен в листинге 7.1.

Листинг 7.1. Программа shellSort.java

```
// shellSort.java
// Сортировка Шелла
// Запуск программы: C>java ShellSortApp
//--------------------------------------------------------------
class ArraySh
```

продолжение ⇗

Листинг 7.1 (*продолжение*)

```
   {
   private long[] theArray;          // Ссылка на массив
   private int nElems;               // Количество элементов
//--------------------------------------------------------------
   public ArraySh(int max)           // Конструктор
      {
      theArray = new long[max];      // Создание массива
      nElems = 0;                    // Пока нет ни одного элемента
      }
//--------------------------------------------------------------
   public void insert(long value)    // Вставка элемента в массив
      {
      theArray[nElems] = value;      // Собственно вставка
      nElems++;                      // Увеличение размера
      }
//--------------------------------------------------------------
   public void display()             // Вывод содержимого массива
      {
      System.out.print("A=");
      for(int j=0; j<nElems; j++)    // Для каждого элемента
      System.out.print(theArray[j] + " ");  // Вывод
      System.out.println("");
   }
//--------------------------------------------------------------
   public void shellSort()
      {
      int inner, outer;
      long temp;
      int h = 1;                     // Вычисление исходного значения h
      while(h <= nElems/3)
         h = h*3 + 1;                // (1, 4, 13, 40, 121, ...)

      while(h>0)                     // Последовательное уменьшение h до 1
         {
                                     // h-сортировка файла
         for(outer=h; outer<nElems; outer++)
            {
            temp = theArray[outer];
            inner = outer;
                                     // Первый подмассив (0, 4, 8)
            while(inner > h-1 && theArray[inner-h] >=  temp)
               {
               theArray[inner] = theArray[inner-h];
               inner -= h;
               }
            theArray[inner] = temp;
            }
         h = (h-1) / 3;              // Уменьшение h
         }
      }
```

```
//--------------------------------------------------------------
   }  // Конец класса ArraySh
////////////////////////////////////////////////////////////////
class ShellSortApp
   {
   public static void main(String[] args)
      {
      int maxSize = 10;             // Размер массива
      ArraySh arr;
      arr = new ArraySh(maxSize);   // Создание массива

      for(int j=0; j<maxSize; j++)  // Заполнение массива
         {                          // случайными числами
         long n = (int)(java.lang.Math.random()*99);
         arr.insert(n);
         }
      arr.display();                // Вывод несортированного массива
      arr.shellSort();              // Сортировка массива по алгоритму Шелла
      arr.display();                // Вывод отсортированного массива
      }
   }  // Конец класса ShellSortApp
```

Метод `main()` создает объект типа `ArraySh` для хранения 10 элементов, заполняет его случайными данными, выводит текущее содержимое, сортирует его методом Шелла и выводит повторно. Пример вывода:

```
A=20 89 6 42 55 59 41 69 75 66
A=6 20 41 42 55 59 66 69 75 89
```

Значение `maxSize` можно увеличить, но не стоит заходить слишком далеко; сортировка 10 000 элементов выполняется за несколько секунд.

Хотя реализация алгоритма Шелла занимает лишь несколько строк, понять ее логику не так уж просто. Чтобы разобраться во всех подробностях происходящего, выполните 10-элементную сортировку в приложении Workshop в пошаговом режиме и сравните выдаваемые сообщения с кодом метода `shellSort()`.

Другие интервальные последовательности

Выбор интервальной последовательности — скорее искусство, нежели наука. В нашем алгоритме до настоящего момента интервальные последовательности генерировались по формуле $h = h \times 3 + 1$, но на практике использовались и другие последовательности с разной степенью успеха. Единственное неизменное требование состоит в том, что уменьшающаяся интервальная последовательность должна завершаться 1, чтобы на последнем проходе выполнялась обычная сортировка методом вставки.

В исходной статье Шелл предложил в качестве исходного интервала значение $N/2$, которое затем просто делилось на 2 при каждом проходе. Таким образом, для $N = 100$ генерировалась последовательность 50, 25, 12, 6, 3, 1. Преимущество такого решения заключается в том, что последовательность не приходится вычислять заранее перед вычислением исходного интервала; достаточно разделить N на 2. Однако такая последовательность не является оптимальной. Хотя для многих

данных она работает эффективнее сортировки методом вставки, в некоторых случаях время выполнения ухудшается до $O(N^2)$ — не лучше, чем у сортировки методом вставки.

В одной из разновидностей этого подхода интервал делится на 2,2 вместо 2. Соответственно для $N = 100$ генерируются числа 45, 20, 9, 4, 1. Такая последовательность существенно лучше простого деления на 2, потому что она позволяет избежать худших случаев с эффективностью $O(N^2)$. Впрочем, в программную реализацию приходится включать дополнительный код, который гарантирует, что последнее значение в последовательности равно 1 независимо от N. По своими показателям эта последовательность сопоставима с последовательностью Кнута, приведенной в листинге.

Еще один вариант выбора последовательности (из книги Фламига; см. приложение Б):

```
if(h < 5)
   h = 1;
else
   h = (5*h-1) / 11;
```

Обычно считается важным, чтобы интервальная последовательность состояла в основном из простых чисел, то есть чтобы числа не имели других делителей, кроме 1. Такое ограничение снижает вероятность повторной обработки элементов, отсортированных при предыдущем проходе. Неэффективность исходной последовательности Шелла $N/2$ отчасти обусловлена именно тем, что она не соответствовала этому правилу.

Возможно, вам удастся изобрести собственную интервальную последовательность, которая работает не хуже (а может, даже лучше) приведенных в этом разделе. Какой бы ни была эта последовательность, ее элементы должны быстро вычисляться, чтобы не замедлять работу алгоритма.

Эффективность сортировки Шелла

До настоящего момента еще никому не удалось теоретически обосновать эффективность сортировки Шелла, если не считать некоторых частных случаев. Оценки, полученные на основании экспериментов, лежат в интервале от O $(N^{3/2})$ до $O(N^{7/6})$.

В таблице 7.2 приведены некоторые оценки эффективности в O-синтаксисе в сравнении как с более медленной сортировкой методом вставки, так и более эффективным алгоритмом быстрой сортировки. Приведены теоретические значения времени для разных значений N. Запись $N^{x/y}$ обозначает корень степени y из N, возведенный в степень x. Например, для $N = 100$ при вычислении $N^{3/2}$ следует извлечь квадратный корень из 100^3; результат равен 1000. Кроме того, запись $(\log N)^2$ означает логарифм N, возведенный в квадрат. Также иногда встречается обозначение $\log^2 N$, но его легко спутать с $\log_2 N$ (логарифмом N по основанию 2).

Для большинства данных более реалистичными оказываются более высокие оценки (такие, как $N^{3/2}$).

Таблица 7.2. Оценки времени выполнения сортировки Шелла

O()	Тип сортировки	10 элементов	100 элементов	1000 элементов	10 000 элементов
N^2	Сортировка методом вставки и др.	100	10 000	1 000 000	100 000 000
$N^{3/2}$	Сортировка Шелла	32	1000	32 000	1 000 000
$N*(\log N)^2$	Сортировка Шелла	10	400	9000	160 000
$N^{5/4}$	Сортировка Шелла	18	316	5600	100 000
$N^{7/6}$	Сортировка Шелла	14	215	3200	46 000
$N*\log N$	Быстрая сортировка и др.	10	200	3000	40 000

Разбиение

Алгоритм быстрой сортировки, который будет описан в следующем разделе, основан на механизме *разбиения*. Впрочем, этот механизм полезен и сам по себе, поэтому ему будет посвящен собственный раздел.

Разбиением данных называется такое разделение на две группы, при котором в одну группу входят все элементы со значением ключа выше заданного порога, а в другую — все элементы со значением ключа ниже заданного порога.

Нетрудно представить себе ситуацию, в которой может возникнуть необходимость в разбиении данных. Допустим, вы хотите разделить данные своих работников на две группы: живущих ближе чем в 15 километрах от офиса и тех, кто живет дальше. Или, допустим, директор школы хочет разделить учеников на две группы: со средним баллом 3,5 и выше и тех, у кого средний балл ниже этого значения.

Приложение Partition Workshop

Приложение Partition Workshop демонстрирует процесс разбиения. На рис. 7.6 изображены 12 столбцов до разбиения, а на рис. 7.7 — те же столбцы после выполнения разбиения.

Горизонтальная линия изображает *опорное значение*, по которому определяется принадлежность элемента к той или иной группе. Элементы с ключом, меньшим опорного значения, размещаются в левой части массива, а элементы с большим (либо равным) ключом — в правой части. (В разделе, посвященном быстрой сортировке, будет показано, что опорное значение может быть ключом реально существующего элемента данных. Пока будем считать его обычным числом.)

Стрелка с подписью `partition` указывает на крайний левый элемент правого (большего) подмассива. Это значение возвращается методом разбиения и может использоваться другими методами, которым нужно знать, где проходит граница разбиения.

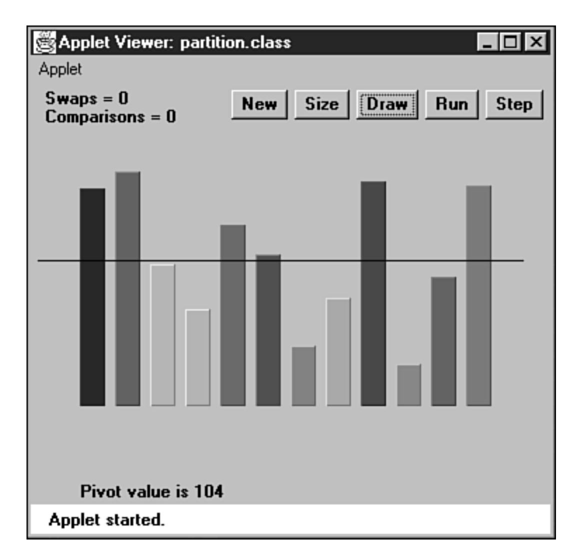


Рис. 7.6. Столбцы до разбиения

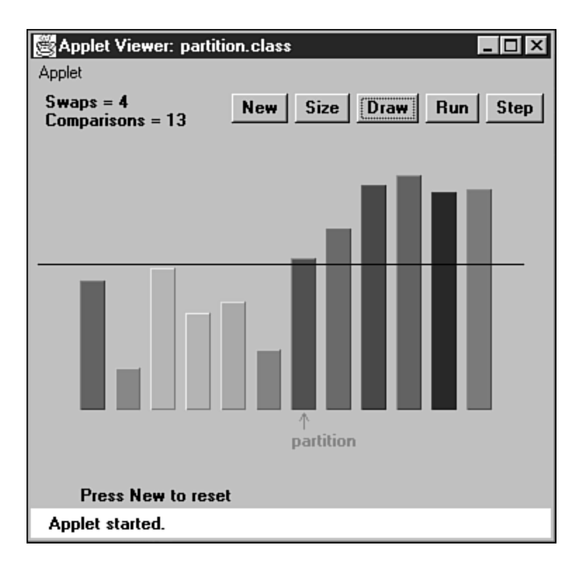


Рис. 7.7. Столбцы после разбиения

Чтобы получить более наглядное представление о процессе разбиения, выберите в приложении Partition Workshop массив с 100 столбцами и нажмите кнопку `Run`. Указатели `leftScan` и `rightScan` продвигаются друг к другу, переставляя столбцы. Когда они встретятся, разбиение будет завершено.

Выбор опорного значения зависит от причины разбиения (например, отбор оценок выше среднего балла 3,5). Для разнообразия приложение Partition Workshop выбирает в качестве опорного значения случайное число (горизонтальная черная линия) при каждом нажатии кнопки `New` или `Size`, но значение никогда не отклоняется слишком далеко от средней высоты столбца.

Разбиение ни в коем случае не обеспечивает сортировки; данные просто делятся на две группы. Тем не менее они становятся более упорядоченными, чем прежде. И как будет показано в следующем разделе, их полная сортировка потребует не столь уж значительных усилий.

Учтите, что разбиение не обладает свойством устойчивости. Иначе говоря, порядок элементов в каждой группе не соответствует исходному порядку следования элементов в массиве. Более того, в результате разбиения часть данных каждой группы обычно переставляется в обратном порядке.

Программа partition.java

Как же выполняется процесс разбиения? Рассмотрим пример кода. В листинге 7.2 приведена программа `partition.java`, содержащая метод `partitionIt()` для разбиения массива.

Листинг 7.2. Программа partition.java

```
// partition.java
// Демонстрация разбиения в массиве
// Запуск программы: C>java PartitionApp
////////////////////////////////////////////////////////////////
class ArrayPar
   {
   private long[] theArray;          // Ссылка на массив
   private int nElems;               // Количество элементов
//--------------------------------------------------------------
   public ArrayPar(int max)          // Конструктор
      {
      theArray = new long[max];      // Создание массива
      nElems = 0;                    // Пока нет ни одного элемента
   }
//--------------------------------------------------------------
   public void insert(long value)    // Вставка элемента в массив
      {
      theArray[nElems] = value;      // Собственно вставка
      nElems++;                      // Увеличение размера
      }
//--------------------------------------------------------------
   public int size()                 // Получение количества элементов
      { return nElems; }
//--------------------------------------------------------------
   public void display()             // Вывод содержимого массива
      {
      System.out.print("A=");
      for(int j=0; j<nElems; j++)    // Для каждого элемента
      System.out.print(theArray[j] + " ");  // Вывод
      System.out.println("");
      }
//--------------------------------------------------------------
```

продолжение ⇨

Листинг 7.1 (*продолжение*)

```
   public int partitionIt(int left, int right, long pivot)
      {
      int leftPtr = left - 1;           // Справа от первого элемента
      int rightPtr = right + 1;         // Слева от опорного элемента
      while(true)
         {
         while(leftPtr < right &&       // Поиск большего элемента
               theArray[++leftPtr] < pivot)
            ;  // (пустое тело цикла)

         while(rightPtr > left &&       // Поиск меньшего элемента
               theArray[--rightPtr] > pivot)
            ;  // (пустое тело цикла)
         if(leftPtr >= rightPtr)        // Если указатели сошлись,
            break;                      // разбиение закончено
         else                           // В противном случае
            swap(leftPtr, rightPtr);    // поменять элементы местами
         }
      return leftPtr;                   // Позиция разбиения
      }
//--------------------------------------------------------------
   public void swap(int dex1, int dex2)  // Перестановка двух элементов
      {
      long temp;
      temp = theArray[dex1];             // A копируется в temp
      theArray[dex1] = theArray[dex2];   // B копируется в A
      theArray[dex2] = temp;             // temp копируется в B
      }
//--------------------------------------------------------------
   }  // Конец класса ArrayPar
////////////////////////////////////////////////////////////////
class PartitionApp
   {
   public static void main(String[] args)
      {
      int maxSize = 16;             // Размер массива
      ArrayPar arr;                 // Ссылка на массив
      arr = new ArrayPar(maxSize);  // Создание массива

      for(int j=0; j<maxSize; j++)  // Заполнение массива
         {                          // случайными числами
         long n = (int)(java.lang.Math.random()*199);
         arr.insert(n);
         }
      arr.display();                // Вывод несортированного массива

      long pivot = 99;              // Опорное значение
      System.out.print("Pivot is " + pivot);
      int size = arr.size();
```

```
                               // Разбиение массива
    int partDex = arr.partitionIt(0, size-1, pivot);

    System.out.println(", Partition is at index " + partDex);
    arr.display();             // Вывод массива после разбиения
    }
```

Метод `main()` создает объект `ArrayPar` для хранения 16 элементов типа `long`. Опорное значение фиксируется равным 99. Метод вставляет в `ArrayPar` 16 случайных значений, выводит их, выполняет разбиение вызовом метода `partitionIt()`, после чего снова выводит содержимое массива. Пример вывода:

```
A=149 192 47 152 159 195 61 66 17 167 118 64 27 80 30 105
Pivot is 99, partition is at index 8
A=30 80 47 27 64 17 61 66 195 167 118 159 152 192 149 105
```

Как видите, разбиение выполнено успешно: первые восемь чисел меньше опорного значения 99, а последние восемь превышают его.

Следует помнить, что процесс разбиения вовсе не обязан делить массив пополам, как в этом примере; это зависит от опорного значения и ключевых значений данных. Количество элементов в одной группе может быть значительно больше, чем в другой.

Алгоритм разбиения

В начале работы алгоритма разбиения два указателя находятся на разных концах массива. (Не забудьте, что термин «указатель» обозначает индекс, однозначно определяющий элемент массива, а не указатель C++.) Левый указатель `leftPtr` перемещается вправо, а правый указатель `rightPtr` перемещается влево. Обратите внимание: `leftPtr` и `rightPtr` в программе `partition.java` соответствуют маркерам `leftScan` и `rightScan` в приложении Partition Workshop.

Строго говоря, `leftPtr` инициализируется позицией слева от первой ячейки, а `rightPtr` — позицией справа от последней ячейки, потому что они соответственно увеличиваются и уменьшаются перед использованием.

Остановка и перестановка

Если `leftPtr` указывает на элемент, меньший опорного значения, он продолжает перемещаться, потому что этот элемент уже находится в правильной группе. Однако при обнаружении элемента, большего опорного, он останавливается. Аналогичным образом, если `rightPtr` указывает на элемент, больший опорного значения, он продолжает перемещаться, а при обнаружении меньшего элемента — останавливается. Процессом проверки управляют два внутренних цикла `while`, для `leftPtr` и для `rightPtr`. Перемещение указателя останавливается из-за выхода из его внутреннего цикла.

Упрощенный код проверки местонахождения элементов выглядит так:

```
while( theArray[++leftPtr] < pivot )   // Поиск большего элемента
   ;  // (пустое тело цикла)
```

```
while( theArray[--rightPtr] > pivot )  // Поиск меньшего элемента
   ;  // (пустое тело цикла)
swap(leftPtr, rightPtr);               // Поменять элементы местами
```

Выполнение первого цикла while прерывается при обнаружении элемента, превышающего опорное значение pivot; выполнение второго цикла — при обнаружении элемента, меньшего pivot. При выходе из обоих циклов как leftPtr, так и rightPtr указывают на элементы, находящиеся в неверных частях массива; эти элементы меняются местами.

После перестановки два указателя продолжают перемещаться. Они снова останавливаются на элементах, находящихся в неверной части массива, и меняют их местами. Все эти операции вложены во внешний цикл while (см. метод partitionIt() в листинге 7.2). Когда два указателя сходятся, процесс разбиения завершен, а выполнение внешнего цикла while также прерывается.

Чтобы проследить за перемещением указателей, запустите приложение Partition Workshop для 100 элементов. Указатели, изображенные синими стрелками, в исходном состоянии находятся у разных концов массива. Они постепенно двигаются навстречу друг другу, останавливаясь и меняя элементы местами. Элементы между указателями еще не были проверены; уже пройденные элементы проверены и находятся в правильном подмассиве. Когда указатели сходятся, разбиение массива завершено.

Работа с нетипичными данными

Если бы мы были уверены в том, что в правой части массива имеется элемент, меньший опорного значения, а в левой части массива — элемент, больший опорного значения, то приведенных ранее упрощенных циклов while было бы вполне достаточно. К сожалению, алгоритм может выполняться для данных, не имеющих столь четкой структуры.

Например, если все данные меньше опорного значения, то переменная leftPtr пройдет по всему массиву в напрасных поисках большего элемента и «выйдет» с правого края с выдачей исключения нарушения границы массива. То же самое произойдет и с rightPtr, если все элементы больше опорного значения.

Чтобы избежать подобных проблем, необходимо включить в циклы while дополнительные проверки выхода за границу массива: leftPtr<right в первом цикле и rightPtr>left — во втором. Эти проверки присутствуют в листинге 7.2.

В разделе, посвященном быстрой сортировке, будет показано, что умно организованный процесс выбора опорного значения позволит обойтись без этих проверок. Объем кода во внутренних циклах всегда следует сводить к минимуму для ускорения работы программы.

Непрочный код

Код циклов while легко испортить неосторожными изменениями. На первый взгляд кажется, что операторы ++ можно исключить из внутренних циклов и использовать их для замены пустых команд в теле цикла. Например, фрагмент

```
while(leftPtr < right && theArray[++leftPtr] < pivot)
   ;  // (пустое тело цикла)
```

можно попытаться заменить следующим фрагментом:

```
while(leftPtr < right && theArray[leftPtr] < pivot)
   ++leftPtr;
```

То же самое делается с другим, внутренним циклом `while`. Эти изменения позволят использовать в качестве начальных значений указателей `left` и `right` вместо менее понятных `left-1` и `right+1`.

Однако в результате таких изменений указатели будут перемещаться только при выполнении условия. А так как для нормальной работы программы они должны перемещаться в любом случае, во внешний цикл `while` придется добавить еще две команды для их смещения. Решение с пустым телом цикла является самым эффективным.

Равенство ключей

Существует еще один неочевидный момент с изменениями в коде `partitionIt()`. Если выполнить `partitionIt()` для массива, элементы которого совпадают с опорным значением, то каждое сравнение будет приводить к перестановке. Казалось бы, перестановка элементов с одинаковыми ключами приводит к лишним тратам времени. Причиной лишних перестановок являются операторы < и >, сравнивающие опорное значение с элементами массива в цикле `while`. Но если вы попытаетесь «оптимизировать» программу, заменив их операторами <= и >=, то одинаковые элементы переставляться действительно не будут, однако `leftPtr` и `rightPtr` при завершении работы алгоритма окажутся на концах массива. Как будет показано в разделе, посвященном быстрой сортировке, весьма желательно, чтобы эти указатели находились в середине массива, а их расположение на концах крайне нежелательно. Таким образом, если метод `partitionIt()` будет использоваться для быстрой сортировки, решение с операторами > и < более эффективно, хотя оно и приводит к некоторым избыточным перестановкам.

Эффективность алгоритма разбиения

Алгоритм разбиения выполняется за время $O(N)$. В этом легко убедиться при помощи приложения Partition Workshop: два указателя начинают движение с разных концов массива и перемещаются друг к другу с более или менее постоянной скоростью, периодически останавливаясь для перестановки элементов. Когда они сходятся, разбиение завершается. Если бы в разбиении участвовало вдвое больше элементов, то указатели двигались бы с той же скоростью, но количество сравниваемых и переставляемых элементов также увеличилось бы вдвое, поэтому процесс занял бы вдвое больше времени. Таким образом, время выполнения пропорционально N.

Говоря более конкретно, для каждого разбиения выполняется $N + 1$ или $N + 2$ сравнений. Каждый элемент обрабатывается и используется в сравнении с тем или иным указателем, что дает N сравнений, но указатели могут «перекрываться» при

заходе друг за друга, поэтому до завершения разбиения будет выполнено одно или два дополнительных разбиения. Количество сравнений не зависит от расположения данных (если не считать неопределенности с одним или двумя дополнительными сравнениями в конце).

Однако количество *перестановок* безусловно зависит от расположения данных. Если данные отсортированы в обратном порядке, а опорное значение делит их количество надвое, то каждую пару элементов придется переставить, а это приведет к $N/2$ перестановок. (Не забывайте, что в приложении Partition Workshop опорное значение выбирается случайным образом, поэтому количество перестановок для элементов, отсортированных в обратном порядке, не всегда в точности равно $N/2$.)

Для случайных данных количество перестановок будет меньше $N/2$ даже в том случае, если ровно половина элементов окажется меньше опорного значения, а другая половина — больше. Дело в том, что некоторые элементы уже будут находиться на положенном месте (малые слева, большие справа). Если опорное значение больше (или меньше) большинства элементов, то количество перестановок дополнительно сократится, поскольку переставлять придется только относительно немногочисленные элементы, бóльшие (или меньшие) опорного значения. В среднем для случайных данных выполняется около половины от максимального количества перестановок.

Хотя количество перестановок меньше количества сравнений, обе величины пропорциональны N. Таким образом, процесс разбиения выполняется за время $O(N)$. Запустив приложение Workshop, вы увидите, что для 12 случайных элементов выполняется около трех перестановок и 14 сравнений, а для 100 случайных элементов — около 25 перестановок и 102 сравнений.

Быстрая сортировка

Бесспорно, быстрая сортировка является самым популярным алгоритмом сортировки, а это вполне понятно: в большинстве ситуаций он выполняется быстрее всех за время $O(N*\log N)$. (Это утверждение справедливо только для внутренней сортировки, выполняемой в памяти; при сортировке данных в файлах на диске другие алгоритмы могут оказаться более эффективными.) Быструю сортировку открыл Ч.Э.Р. Хоар в 1962 году.

Чтобы понять суть быстрой сортировки, необходимо знать алгоритм разбиения, описанный в предыдущем разделе. По сути, алгоритм быстрой сортировки разбивает массив на два подмассива, а затем рекурсивно вызывает себя для выполнения быстрой сортировки каждого из этих подмассивов. Впрочем, в эту базовую схему можно внести ряд усовершенствований, относящихся к выбору опорного значения и сортировке малых подмассивов. Эти усовершенствования будут описаны после рассмотрения простой версии основного алгоритма.

Трудно понять, *что* происходит при быстрой сортировке, не зная, *как* это делается, поэтому в этом разделе обычный порядок изложения изменяется: сначала мы приведем код быстрой сортировки на языке Java, а затем перейдем к вспомогательному приложению QuickSort1 Workshop.

Алгоритм быстрой сортировки

Код базового рекурсивного метода быстрой сортировки относительно прост. Пример:

```
public void recQuickSort(int left, int right)
   {
   if(right-left <= 0)        // Если размер равен 1,
      return;                 // массив отсортирован.
   else                       // Для размера 2 и более
      {
                              // Разбиение диапазона
      int partition = partitionIt(left, right);
      recQuickSort(left, partition-1);   // Сортировка левой части
      recQuickSort(partition+1, right);  // Сортировка правой части
      }
   }
```

Алгоритм состоит из трех основных шагов:

1. Массив или подмассив разбивается на две группы: левую (с меньшими ключами) и правую (с большими ключами).
2. Рекурсивный вызов метода для сортировки левой группы.
3. Рекурсивный вызов метода для сортировки правой группы.

После разбиения все элементы левого подмассива меньше элементов правого подмассива. Если отсортировать левый подмассив и правый подмассив, то весь массив будет упорядочен. Как отсортировать эти подмассивы? Посредством рекурсивных вызовов.

Аргументы метода `recQuickSort()` определяют левую и правую границы массива (или подмассива), который должен сортироваться при вызове. Метод сначала проверяет, не состоит ли массив только из одного элемента. Если условие выполняется, значит, массив уже отсортирован, и метод немедленно возвращает управление. Это базовое ограничение процесса рекурсии.

Если массив состоит из двух и более ячеек, то алгоритм вызывает метод `partitionIt()` (см. предыдущий раздел) для его разбиения. Метод возвращает *индекс разбиения* — индекс левого элемента правого подмассива (с бо́льшими значениями ключей). Это значение отмечает положение границы между подмассивами. Ситуация изображена на рис. 7.8.



Рис. 7.8. Сортировка подмассивов в результате рекурсивных вызовов

После разбиения массива метод `recQuickSort()` вызывает себя рекурсивно: для левой части массива (от `left` до `partition-1`) и для правой части массива (от `partition+1` до `right`). Обратите внимание: элемент данных с индексом `partition` не включается ни в один из рекурсивных вызовов. Почему? Разве его не нужно сортировать? Ответ на этот вопрос кроется в выборе опорного значения.

Выбор опорного значения

Какое опорное значение должен использовать метод `partitionIt()`? Несколько разумных идей:

- Опорное значение должно быть ключом существующего элемента; этот элемент называется опорным.
- Опорный элемент выбирается более или менее произвольно. Для простоты мы всегда будем выбирать элемент, находящийся у правого края массива.
- После разбиения опорный элемент, вставленный на границе между левым и правым подмассивом, всегда находится в своей окончательной позиции сортировки.

Последний пункт кажется неочевидным, но вспомните: так как ключ опорного элемента используется для разбиения массива, после его выполнения в левом подмассиве собираются элементы, меньшие опорного, а в правом — элементы, большие опорного. Опорный элемент изначально находится у правого края, но если бы он был каким-то образом перемещен между двумя подмассивами, то он бы оказался на правильном месте, то есть в своей итоговой позиции сортировки. На рис. 7.9 показано, как выглядит схема разбиения с опорным элементом, имеющим ключ 36.

В действительности дело обстоит сложнее — массив нельзя «раздвинуть», как это сделано на рисунке. Как же переместить опорный элемент в положенное место?

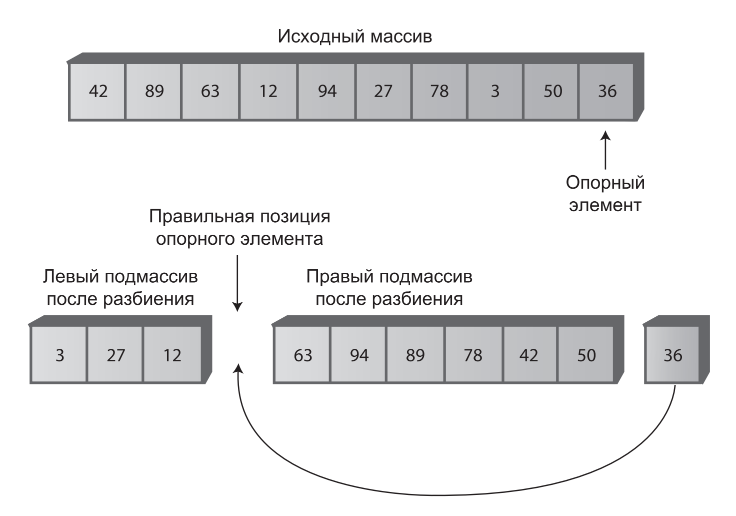
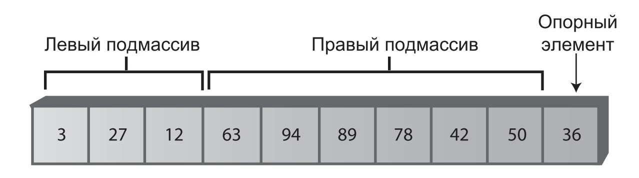


Рис. 7.9. Опорный элемент и подмассивы

Конечно, можно сдвинуть все элементы правого подмассива на одну ячейку, чтобы освободить место. Тем не менее такое решение неэффективно, но есть и другие способы. Вспомните, что все элементы правого подмассива, пусть и бóльшие опорного значения, еще не отсортированы, поэтому их перемещение в правом подмассиве ни на что не повлияет. Следовательно, для упрощения вставки опорного элемента в нужной позиции мы можем просто поменять местами опорный элемент (36) с левым элементом правого подмассива (63). В результате перестановки опорный элемент занимает положенное место в позиции между левой и правой группами. Элемент 63 перемещается к правому краю, но поскольку он остается в правой (большей) группе, разбиение при этом не нарушается. Ситуация показана на рис. 7.10.



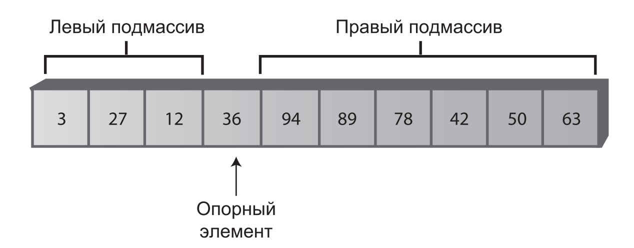


Рис. 7.10. Перестановка опорного элемента

После перестановки с элементом, находящимся в позиции разбиения, опорный элемент занимает свое окончательное место. Все последующие действия будут выполняться по одну или другую сторону от него, но сам опорный элемент двигаться не будет (более того, алгоритм даже не будет обращаться к нему).

В первой версии выбора опорного элемента в методе `recQuickSort()` значение будет передаваться явно в аргументе `partitionIt()`. Первая версия выглядит так:

```
public void recQuickSort(int left, int right)
   {
   if(right-left <= 0)        // Если размер <= 1,
      return;                 // массив отсортирован.
   else                       // Для размера 2 и более
      {
      long pivot = theArray[right];       // Крайний правый элемент
                                          // Разбиение диапазона
      int partition = partitionIt(left, right, pivot);
      recQuickSort(left, partition-1);   // Сортировка левой части
      recQuickSort(partition+1, right);  // Сортировка правой части
      }
   }
```

При использовании схемы с выбором крайнего правого элемента массива в качестве опорного необходимо изменить метод `partitionIt()`, чтобы крайний правый элемент был исключен из процесса разбиения. Ведь мы уже знаем, что после завершения процесса он должен находиться между двумя группами. Кроме того, после завершения разбиения необходимо поменять местами опорный элемент на правом конце и элемент в позиции разбиения. В листинге 7.3 приведена программа `quickSort1.java`, в которой реализованы все эти возможности.

Листинг 7.3. Программа quickSort1.java

```
// quickSort1.java
// Простая версия быстрой сортировки
// Запуск программы: C>java QuickSort1App
////////////////////////////////////////////////////////////////
class ArrayIns
   {
   private long[] theArray;          // Ссылка на массив
   private int nElems;               // Количество элементов
//--------------------------------------------------------------
   public ArrayIns(int max)          // Конструктор
      {
      theArray = new long[max];      // Создание массива
      nElems = 0;                    // Пока нет ни одного элемента
      }
//--------------------------------------------------------------
   public void insert(long value)    // Вставка элемента в массив
      {
      theArray[nElems] = value;      // Собственно вставка
      nElems++;                      // Увеличение размера
      }
```

```
//--------------------------------------------------------------
   public void display()            // Вывод содержимого массива
      {
      System.out.print("A=");
      for(int j=0; j<nElems; j++)    // Для каждого элемента
      System.out.print(theArray[j] + " ");  // Вывод
      System.out.println("");
      }
//--------------------------------------------------------------
   public void quickSort()
      {
      recQuickSort(0, nElems-1);
      }
//--------------------------------------------------------------
   public void recQuickSort(int left, int right)
      {
      if(right-left <= 0)              // Если размер <= 1,
         return;                       //    массив отсортирован
      else                             // Для размера 2 и более
         {
         long pivot = theArray[right];      // Крайний правый элемент
                                            // Разбиение диапазона
         int partition = partitionIt(left, right, pivot);
         recQuickSort(left, partition-1);   // Сортировка левой части
         recQuickSort(partition+1, right);  // Сортировка правой части
         }
      }
//--------------------------------------------------------------
   public int partitionIt(int left, int right, long pivot)
      {
      int leftPtr = left-1;           // Левая граница (после ++)
      int rightPtr = right;           // Правая граница-1 (after --)
      while(true)
         {                            // Поиск большего элемента
         while( theArray[++leftPtr] < pivot )
            ;  // (nop)
                                      // Поиск меньшего элемента
         while(rightPtr > 0 && theArray[--rightPtr] > pivot)
            ;  // (nop)

         if(leftPtr >= rightPtr)      // Если указатели сошлись,
            break;                    // разбиение закончено.
         else                         // В противном случае
            swap(leftPtr, rightPtr);  // поменять элементы местами.
         }
      swap(leftPtr, right);           // Перестановка опорного элемента
      return leftPtr;                 // Возврат позиции опорного элемента
      }
//--------------------------------------------------------------
```

продолжение ⇨

Листинг 7.3 (*продолжение*)

```
   public void swap(int dex1, int dex2)  // Перестановка двух элементов
      {
      long temp;
      temp = theArray[dex1];             // A копируется в temp
      theArray[dex1] = theArray[dex2];   // B копируется в A
      theArray[dex2] = temp;             // temp копируется в B
      }
//--------------------------------------------------------------
   }  // Конец класса ArrayIns
////////////////////////////////////////////////////////////////
class QuickSort1App
   {
   public static void main(String[] args)
      {
      int maxSize = 16;             // Размер массива
      ArrayIns arr;
      arr = new ArrayIns(maxSize);  // Создание массива
      for(int j=0; j<maxSize; j++)  // Заполнение массива
         {                          // случайными числами.
         long n = (int)(java.lang.Math.random()*99);
         arr.insert(n);
         }
      arr.display();                // Вывод элементов
      arr.quickSort();              // Быстрая сортировка
      arr.display();                // Повторный вывод элементов
      }
   }  // Конец класса QuickSort1App
```

Метод `main()` создает объект типа `ArrayIns`, вставляет в массив 16 случайных элементов типа `long`, сортирует методом `quickSort()` и выводит результат. Результат выполнения выглядит примерно так:

```
A=69 0 70 6 38 38 24 56 44 26 73 77 30 45 97 65
A=0 6 24 26 30 38 38 44 45 56 65 69 70 73 77 97
```

У кода метода `partitionIt()` есть одна интересная особенность: нам удалось обойтись без проверки конца массива в первом внутреннем цикле `while`. В приводившейся ранее версии метода `partitionIt()` программы `partition.java` в листинге 7.2 эта проверка выглядела так:

```
leftPtr < right
```

Проверка не позволяла `leftPtr` выйти за правую границу массива, если в нем не было ни одного элемента, большего `pivot`. Почему этот тест оказался лишним? Потому что в качестве опорного выбирается крайний правый элемент, а `leftPtr` всегда будет останавливаться на нем. Впрочем, для `rightPtr` во втором цикле `while` такая проверка все еще необходима. (Позднее вы увидите, как избавиться и от нее.)

Выбор крайнего правого элемента в качестве опорного не является абсолютно случайным; он ускоряет выполнение кода за счет исключения ненужной проверки. Выбор опорного значения в другой позиции таким преимуществом не обладает.

Приложение QuickSort1 Workshop

На этой стадии вы об алгоритме быстрой сортировки уже достаточно знаете, чтобы разобраться во всех тонкостях приложения QuickSort1 Workshop.

Общая картина

Чтобы получить общее представление о работе приложения, выберите режим сортировки 100 элементов кнопкой `Size`, а затем нажмите кнопку `Run`. Примерный вид приложения после завершения сортировки показан на рис. 7.11.

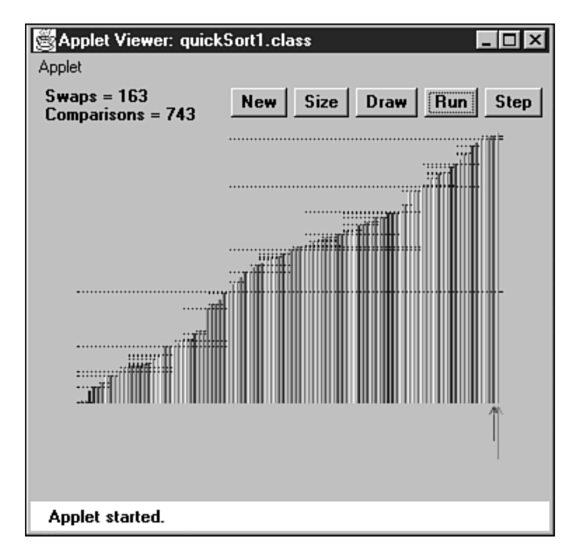


Рис. 7.11. Приложение QuickSort1 Workshop со 100 элементами

Проследите за тем, как алгоритм делит массив на две части, а затем сортирует каждую из частей, разделяя ее на две части — и так далее, с созданием подмассивов постоянно уменьшающихся размеров.

После завершения процесса сортировки пунктирными линиями помечаются все отсортированные подмассивы. Горизонтальная протяженность линии показывает, какие элементы входили в подмассив, а вертикальная определяет опорное значение (высота столбца). Общая длина всех линий является мерой объема работы, выполняемой алгоритмом для сортировки массива; мы еще вернемся к этой теме. Выше и ниже каждой пунктирной линии (кроме самых коротких) должны находиться другие линии (возможно, отделенные другими, более короткими линиями), причем суммарная длина этих линий равна длине исходной линии (без одного элемента). Линии обозначают две группы, на которые делится каждый подмассив.

Подробное описание

Чтобы более тщательно разобраться в том, как выполняется быстрая сортировка, переключитесь на массив с 12 элементами в приложении QuickSort1 Workshop и выполните процесс сортировки в пошаговом режиме. Вы увидите, что опорное значение соответствует высоте столбца у правого края массива, а алгоритм выполняет разбиение, меняет местами опорный элемент с элементом между двумя группами, сортирует меньшую группу (с использованием многочисленных рекурсивных вызовов), а затем переходит к сортировке большей группы.

На рис. 7.12 представлена вся последовательность сортировки 12 элементов. Горизонтальные скобки под массивами показывают, какой подмассив подвергается разбиению на каждом шаге, а числа в кружках — в каком порядке создаются группы. Перемещение опорного элемента на положенное место обозначается пунктирной линией со стрелкой. Итоговая позиция опорного элемента выделена пунктиром — это означает, что ячейка содержит уже отсортированный элемент и в будущем изменяться не будет. Горизонтальная скобка под отдельной ячейкой (шаги 5, 6, 7, 11 и 12) обозначает вызовы `recQuickSort()` для базовых ограничений; такие вызовы немедленно возвращают управление.

В некоторых случаях (шаги 4 и 10) опорный элемент оказывается в своей исходной позиции на правом крае сортируемого массива. В подобной ситуации остается отсортировать только один подмассив: тот, что расположен слева от опорного элемента. Второй (правый) подмассив отсутствует.

Действия, показанные на рис. 7.12, выполняются на разных уровнях рекурсии — см. табл. 7.3. Исходный вызов `recQuickSort()` из `main()` находится на первом уровне, вызов из `recQuickSort()` двух новых экземпляров самого себя — на втором уровне, вызов еще четырех экземпляров из этих двух — на третьем и т. д.

Таблица 7.3. Уровни рекурсии на рис. 7.12

Шаг	Уровень рекурсии
1	1
2, 8	2
3, 7, 9	3
4, 10	4
5, 6, 11	5

Порядок выполнения разбиений соответствует номерам шагов, а не глубине. Не стоит полагать, что сначала выполняются все разбиения первого уровня, потом все разбиения второго уровня и т. д. Сначала обрабатываются все левые группы на всех уровнях, и только потом происходит переход к правым группам.

Теоретически на четвертом уровне должно быть 8 шагов, а на пятом — 16, но в нашем небольшом массиве все элементы были исчерпаны еще до того, как возникла необходимость в них.

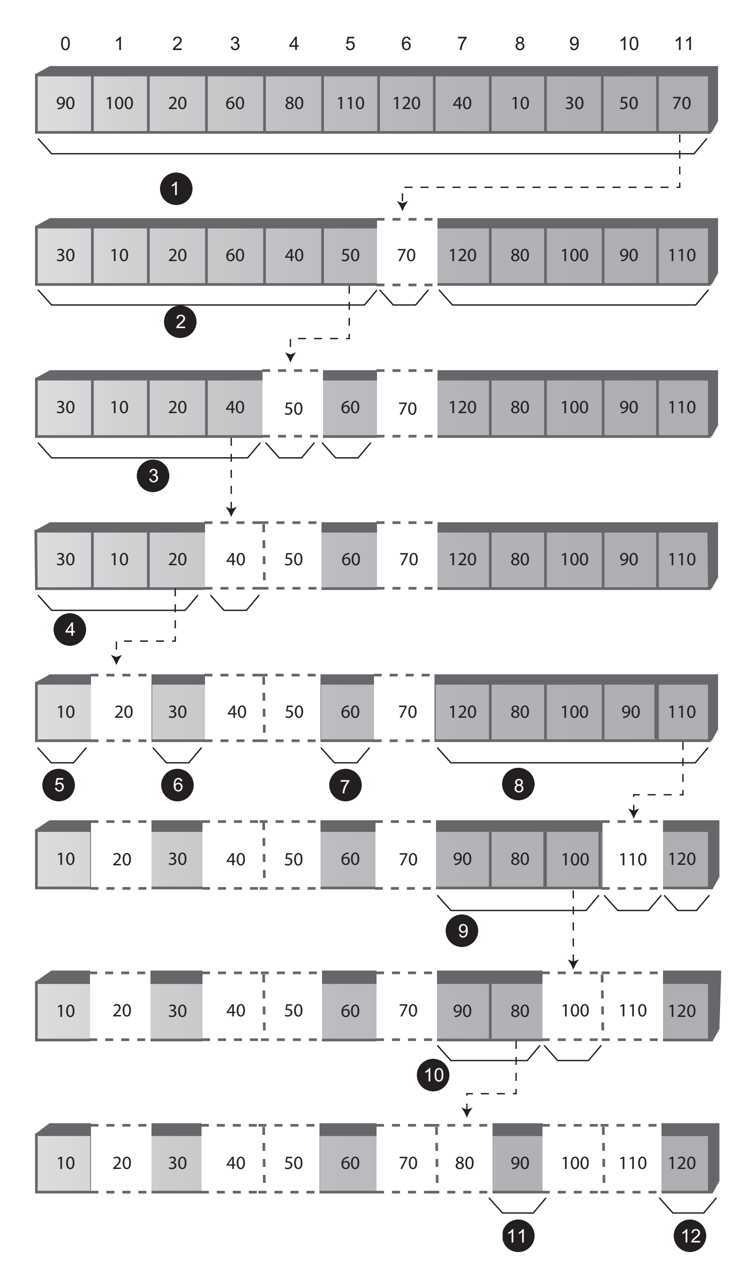


Рис. 7.12. Процесс быстрой сортировки

Количество уровней в таблице показывает, что при 12 элементах в машинном стеке резервируется место для 5 наборов аргументов и возвращаемых значений: по одному для каждого уровня рекурсии. Как будет показано далее, это несколько

больше логарифма количества элементов по основанию 2 ($\log_2 N$). Размер машинного стека определяется конкретной системой. Сортировка очень больших массивов данных с применением рекурсивных процедур может привести к переполнению стека и порче содержимого памяти.

На что следует обратить внимание

При запуске приложения QuickSort1 Workshop следует обратить внимание на некоторые подробности. У этой версии алгоритма быстрой сортировки не возникает ни малейших проблем с небольшими подмассивами, состоящими из двух или трех элементов: `leftScan` и `rightScan` просто сразу сходятся, и сортировка на этом заканчивается. По этой причине нам не нужно создавать отдельную схему сортировки для малых подмассивов. (Хотя, как будет показано позднее, отдельная обработка малых подмассивов может иметь свои преимущества.)

В конце каждого прохода переменная `leftScan` указывает на левый элемент правого подмассива. Далее, как мы видели, опорный элемент меняется местами с этим элементом и занимает свое законное место. При выполнении шагов 3 и 9 на рис. 7.12 переменная `leftScan` в конечном итоге указывает на сам опорный элемент, так что фактически перестановка ничего не меняет. На первый взгляд кажется, что перестановка получается лишней, а переменной `leftScan` стоило бы остановиться на один элемент раньше. Однако очень важно, чтобы переменная `leftScan` прошла весь путь до опорного элемента; в противном случае перестановка нарушит сортировку опорного элемента и подмассива.

Не забудьте, что переменные `leftScan` и `rightScan` начинают перемещение с позиций `left-1` и `right`. В приложении они могут выглядеть странно, особенно если значение `left` равно 0; в этом случае исходное значение `leftScan` равно –1. Аналогичным образом `rightScan` изначально указывает на опорный элемент, который не включен в процесс разбиения. Указатели в исходном состоянии находятся за пределами разбиваемого подмассива, потому что перед первым использованием с ними будут выполнены операции увеличения и уменьшения соответственно. В приложении диапазоны обозначаются числами в круглых скобках; например, (2–5) обозначает подмассив от индекса 2 до индекса 5. В некоторых сообщениях диапазон может быть отрицательным, то есть идти от большего числа к меньшему — как, например, в сообщении *Array partitioned; left (7–6), right (8–8)*. Диапазон (8–8) представляет одну ячейку (8), но что такое (7–6)? Этот диапазон реально не существует; в записи просто отражены значения аргументов `recQuickSort()` `left` и `right` при вызове метода. Соответствующий фрагмент кода:

```
int partition = partitionIt(left, right, pivot);
recQuickSort(left, partition-1);   // Сортировка левой части
recQuickSort(partition+1, right);  // Сортировка правой части
```

Например, если `partitionIt()` вызывается с аргументами `left = 7` и `right = 8` и возвращает 7 в качестве индекса разбиения, то при первом рекурсивном вызове `recQuickSort()` будет указан диапазон (7–6), а при втором — диапазон (8–8). Это нормально. Базовое ограничение `recQuickSort()` активизируется при размере массива, меньшем либо равном 1, поэтому для отрицательных диапазонов тоже проис-

ходит немедленный возврат управления. Отрицательные диапазоны не показаны на рис. 7.12, хотя они тоже порождают (кратковременные) вызовы `recQuickSort()`.

Вырожденное быстродействие $O(N^2)$

Воспользовавшись приложением QuickSort1 Workshop для 100 элементов, отсортированных в обратном порядке, вы заметите, что алгоритм работает намного медленнее, а в приложении выводится намного больше пунктирных горизонтальных линий, то есть разбиению подвергается большее количество подмассивов. Что происходит?

Проблема возникает из-за выбора опорного значения. В идеале опорное значение должно быть медианой сортируемых элементов. Иначе говоря, половина элементов должна быть больше опорного значения, а другая половина — меньше его. Это приведет к тому, что массив будет разбиваться на два подмассива равных размеров. Ситуация с двумя подмассивами равных размеров оптимальна для алгоритма быстрой сортировки. Если алгоритму приходится сортировать два массива разных размеров, его эффективность ухудшается, поскольку больший массив приходится разбивать большее количество раз.

Самая худшая ситуация — разбиение массива из N элементов на два подмассива, из 1 и $N - 1$ элементов. (Такое деление также встречается на шагах 3 и 9 на рис. 7.12.) Если такое деление на 1 и $N - 1$ происходит с каждым разбиением, то на каждый элемент потребуется свой шаг разбиения. Именно это происходит с данными, отсортированными в обратном порядке: во всех подмассивах опорный элемент является наименьшим, поэтому при каждом разбиении первый подмассив содержит $N - 1$ элементов, а во второй входит только опорный элемент.

Чтобы понаблюдать за этой неприятной ситуацией, выполните пошаговую сортировку в приложении QuickSort1 Workshop для 12 элементов, отсортированных в обратном порядке. Обратите внимание, насколько больше требуется шагов по сравнению со случайными данными. В данной ситуации преимущество от процесса разбиения теряется, а эффективность алгоритма вырождается до $O(N^2)$.

Помимо неэффективности, быстрая сортировка за время $O(N^2)$ создает еще одну проблему. При увеличении количества разбиений соответственно возрастает и количество рекурсивных вызовов. Каждый вызов занимает место в машинном стеке. Если вызовов окажется слишком много, возможно переполнение стека и парализация системы.

Подведем итог: в приложении QuickSort1 в качестве опорного выбирается крайний правый элемент. Для случайных данных такой выбор неплох, потому что опорное значение редко оказывается слишком близким к концу массива. Тем не менее, если данные отсортированы (в прямом или обратном порядке), выбор опорного элемента с края массива становится нежелательным. Нельзя ли усовершенствовать наш подход к выбору опорного значения?

Определение медианы по трем точкам

Существует много схем выбора опорного значения. Такая схема должна быть простой, но при этом с большой вероятностью избегать выбора наибольшего или наименьшего значения. Случайный выбор элемента прост, но как мы уже видели, не всегда приводит к хорошему результату. Конечно, можно проанализировать все элементы и вычислить медиану. Выбор будет идеальным, но сам процесс вряд ли можно признать практичным — он займет больше времени, чем сама сортировка.

В одном из компромиссных решений медиана определяется по первому, последнему и среднему элементам в массиве. Полученное значение используется в качестве опорного. Выбор медианы первого, последнего и среднего элементов называется определением *медианы по трем точкам* (рис. 7.13).

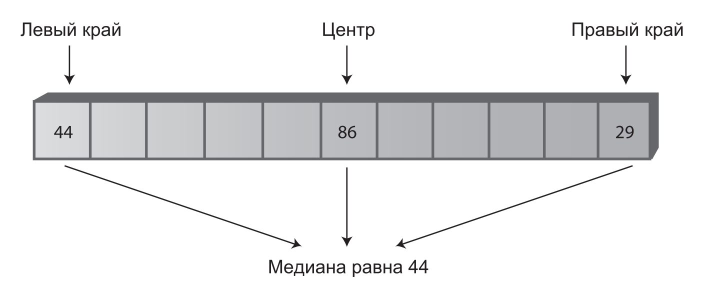
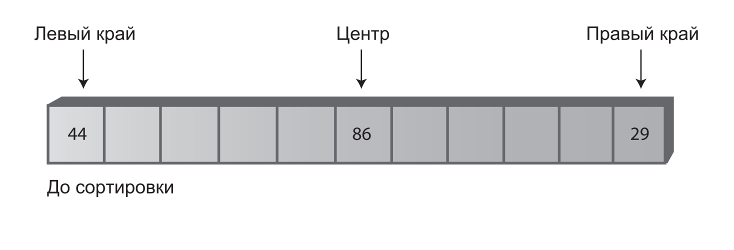


Рис. 7.13. Определение медианы по трем точкам

Конечно, определение медианы по трем точкам происходит намного быстрее, чем определение по всем элементам массива. При этом оно успешно избегает выбора наибольшего или наименьшего элемента в тех случаях, когда данные уже отсортированы в прямом или обратном порядке. Вероятно, в некоторых патологических случаях эта схема тоже будет плохо работать, но в обычной ситуации она позволяет быстро и эффективно выбрать опорное значение.

Кроме эффективности выбора опорного значения, у определения медианы по трем точкам имеется дополнительное преимущество: проверку `rightPtr>left` можно включить во второй внутренний цикл `while`, что приведет к небольшому ускорению работы алгоритма. Как такое возможно? Мы воспользуемся определением медианы по трем точкам не только для выбора опорного значения, но и для сортировки трех элементов, использованных в процессе выбора. Операция показана на рис. 7.14.

После сортировки трех элементов и выбора медианы в качестве опорного значения точно известно, что элемент у левого края подмассива меньше опорного значения (либо равен ему), а элемент у правого края больше опорного значения (или равен ему). Это означает, что индексы `leftPtr` и `rightPtr` не могут выйти за правую и левую границы массива соответственно, даже если убрать проверки `leftPtr>right` и `rightPtr<left`. (Указатель остановится в предположении, что элементы необходимо поменять местами, но обнаружит, что произошло схождение, а разбиение завершено.) Значения `left` и `right` выполняют функции *стражей* (sentinels), ограничивающих `leftPtr` и `rightPtr` действительными значениями для данного массива.



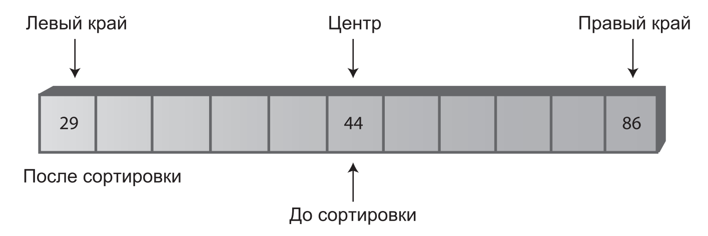


Рис. 7.14. Сортировка левого, среднего и правого элементов

У определения медианы по трем точкам имеется еще одно второстепенное преимущество: после сортировки трех элементов процессу разбиения уже не придется проверять их заново. Разбиение можно начать с позиций `left+1` и `right-1`, потому что элементы `left` и `right` фактически уже прошли его. Мы знаем, что элемент `left` принадлежит правильному подмассиву, потому что он находится слева и его значение меньше опорного, а элемент `right` — потому что он находится справа, а его значение больше опорного.

Таким образом, разбиение по медиане, определенной по трем точкам, не только избегает снижения эффективности до $O(N^2)$ с уже отсортированными данными, но и способствует ускорению внутренних циклов алгоритма разбиения, а также незначительно сокращает количество обрабатываемых элементов.

Программа quickSort2.java

В листинге 7.4 приведена программа `quickSort2.java`, в которой реализовано определение медианы по трем точкам. Для сортировки левого, среднего и правого элементов используется отдельный метод `medianOf3()`. Метод возвращает опорное значение, которое затем передается методу `partitionIt()`.

Листинг 7.4. Программа quickSort2.java

```
// quickSort2.java
// Быстрая сортировка с определением медианы по трем точкам
// Запуск программы: C>java QuickSort2App
////////////////////////////////////////////////////////////////
class ArrayIns
   {
   private long[] theArray;          // Ссылка на массив
   private int nElems;               // Количество элементов
```

продолжение ⇗

Листинг 7.4 (*продолжение*)

```
//--------------------------------------------------------------
   public ArrayIns(int max)          // Конструктор
      {
      theArray = new long[max];      // Создание массива
      nElems = 0;                    // Пока нет ни одного элемента
      }
//--------------------------------------------------------------
   public void insert(long value)    // Вставка элемента в массив
      {
      theArray[nElems] = value;      // Собственно вставка
      nElems++;                      // Увеличение размера
      }
//--------------------------------------------------------------
   public void display()             // Вывод содержимого массива
      {
      System.out.print("A=");
      for(int j=0; j<nElems; j++)    // Для каждого элемента
         System.out.print(theArray[j] + " ");  // Вывод
      System.out.println("");
      }
//--------------------------------------------------------------
   public void quickSort()
      {
      recQuickSort(0, nElems-1);
      }
//--------------------------------------------------------------
   public void recQuickSort(int left, int right)
      {
      int size = right-left+1;
      if(size <= 3)               // Ручная сортировка при малом размере
         manualSort(left, right);
      else                        // Быстрая сортировка при большом размере
         {
         long median = medianOf3(left, right);
         int partition = partitionIt(left, right, median);
         recQuickSort(left, partition-1);
         recQuickSort(partition+1, right);
         }
      }
//--------------------------------------------------------------
   public long medianOf3(int left, int right)
      {
      int center = (left+right)/2;
                                    // Упорядочение left и center
      if( theArray[left] > theArray[center] )
         swap(left, center);
                                    // Упорядочение left и right
      if( theArray[left] > theArray[right] )
         swap(left, right);
```

```
                              // Упорядочение center и right
      if( theArray[center] > theArray[right] )
         swap(center, right);

      swap(center, right-1);         // Размещение медианы на правом краю
      return theArray[right-1];      // Метод возвращает медиану
      }
//-----------------------------------------------------------------
   public void swap(int dex1, int dex2)  // Перестановка двух элементов
      {
      long temp = theArray[dex1];        // A копируется в temp
      theArray[dex1] = theArray[dex2];   // B копируется в A
      theArray[dex2] = temp;             // temp копируется в B
      }
//-----------------------------------------------------------------
   public int partitionIt(int left, int right, long pivot)
      {
      int leftPtr = left;              // Справа от первого элемента
      int rightPtr = right - 1;        // Слева от опорного элемента

      while(true)
         {
         while( theArray[++leftPtr] < pivot )  // Поиск большего элемента
            ;                                  // (пустое тело цикла)
         while( theArray[--rightPtr] > pivot ) // Поиск меньшего элемента
            ;                                  // (пустое тело цикла)
         if(leftPtr >= rightPtr)       // Если указатели сошлись,
            break;                     // разбиение закончено
         else                          // В противном случае
            swap(leftPtr, rightPtr);   // поменять элементы местами
         }
      swap(leftPtr, right-1);          // Восстановление опорного элемента
      return leftPtr;                  // Позиция разбиения
      }
//-----------------------------------------------------------------
   public void manualSort(int left, int right)
      {
      int size = right-left+1;
      if(size <= 1)
         return;         // Сортировка не требуется
      if(size == 2)
         {               // 2-сортировка left и right
         if( theArray[left] > theArray[right] )
            swap(left, right);
         return;
         }
      else               // Размер равен 3
         {               // 3-сортировка left, center и right
         if( theArray[left] > theArray[right-1] )
```

продолжение ⇨

Листинг 7.4 (*продолжение*)

```
            swap(left, right-1);                // left, center
         if( theArray[left] > theArray[right] )
            swap(left, right);                  // left, right
         if( theArray[right-1] > theArray[right] )
            swap(right-1, right);               // center, right
         }
      }
//--------------------------------------------------------------
   }  // Конец класса ArrayIns
////////////////////////////////////////////////////////////////
class QuickSort2App
   {
   public static void main(String[] args)
      {
      int maxSize = 16;             // Размер массива
      ArrayIns arr;                 // Ссылка на массив
      arr = new ArrayIns(maxSize);  // Создание массива
      for(int j=0; j<maxSize; j++)  // Заполнение массива
         {                          // случайными числами
         long n = (int)(java.lang.Math.random()*99);
         arr.insert(n);
         }
      arr.display();                // Вывод элементов
      arr.quickSort();              // Быстрая сортировка
      arr.display();                // Повторный вывод
      }
   }  // Конец класса QuickSort2App
```

Для сортировки подмассивов из трех и менее элементов в программе используется новый метод `manualSort()`. Если массив состоит из одной (и менее) ячейки, метод немедленно возвращает управление; для массива размера 2 элементы переставляются местами при необходимости; для массива размера 3 происходит сортировка трех элементов. Метод `recQuickSort()` не может использоваться для сортировки диапазонов с двумя или тремя элементами, потому что разбиение с медианой требует минимум четырех ячеек.

Метод `main()` и выходные данные `quickSort2.java` выглядят так же, как в программе `quickSort1.java`.

Приложение QuickSort2 Workshop

Приложение QuickSort2 Workshop демонстрирует работу алгоритма быстрой сортировки с определением медианы по трем точкам. В целом приложение аналогично QuickSort1 Workshop, но его работа начинается с сортировки первого, центрального и последнего элементов каждого подмассива, с выбором их медианы в качестве опорного значения. По крайней мере приложение делает это, если размер массива больше трех. Если размер составляет два или три элемента, приложение сортирует их «вручную» без разбиений и рекурсивных вызовов.

Обратите внимание на кардинальное улучшение быстродействия при сортировке 100 элементов, отсортированных в обратном порядке. Массивы разбиваются не на 1 и $N - 1$ ячейку, а приблизительно пополам.

Если не считать этого усовершенствования для упорядоченных данных, приложение QuickSort2 Workshop выдает те же результаты, что и QuickSort1. При сортировке случайных данных никакого выигрыша по скорости нет; преимущества проявляются только при сортировке упорядоченных данных.

Обработка малых подмассивов

Если вы используете метод определения медианы по трем точкам, то алгоритм быстрой сортировки не будет работать с подмассивами из трех и менее элементов. Число 3 в данном случае называется *порогом отсечения*. В приведенных ранее примерах подмассивы из двух или трех элементов сортировались вручную. Нет ли лучшего способа?

Применение сортировки методом вставки для малых подмассивов

Другой вариант обработки малых подмассивов основан на применении сортировки методом вставки. В этом случае отсечение не ограничивается значением 3 — вы можете установить его равным 10, 20 или любому другому числу. Интересно поэкспериментировать с разными величинами отсечения и определить, в каком случае достигается лучшее быстродействие. Кнут (см. приложение Б) рекомендует использовать отсечение 9, однако оптимальное значение зависит от компьютера, операционной системы, компилятора (или интерпретатора) и т. д.

В программе `quickSort3.java` (листинг 7.5) сортировка методом вставки используется для обработки подмассивов, содержащих менее 10 элементов.

Листинг 7.5. Программа quickSort3.java

```
// quickSort3.java
// Быстрая сортировка с применением сортировки методом вставки
// Запуск программы: C>java QuickSort3App
////////////////////////////////////////////////////////////////
class ArrayIns
   {
   private long[] theArray;          // Ссылка на массив
   private int nElems;               // Количество элементов
//--------------------------------------------------------------
   public ArrayIns(int max)          // Конструктор
      {
      theArray = new long[max];      // Создание массива
      nElems = 0;                    // Пока нет ни одного элемента
      }
//--------------------------------------------------------------
```

продолжение ⇗

Листинг 7.5 (*продолжение*)

```
   public void insert(long value)    // Вставка элемента в массив
      {
      theArray[nElems] = value;      // Собственно вставка
      nElems++;                      // Увеличение размера
      }
//-------------------------------------------------------------
   public void display()             // Вывод содержимого массива
      {
      System.out.print("A=");
      for(int j=0; j<nElems; j++)    // Для каждого элемента
         System.out.print(theArray[j] + " ");  // Вывод
      System.out.println("");
      }
//-------------------------------------------------------------
   public void quickSort()
      {
      recQuickSort(0, nElems-1);
      // insertionSort(0, nElems-1); // Другой вариант
      }
//-------------------------------------------------------------
   public void recQuickSort(int left, int right)
      {
      int size = right-left+1;
      if(size < 10)                   // Сортировка методом вставки
         insertionSort(left, right);  // при малом размере
      else                            // Быстрая сортировка
         {                            // при большом размере
         long median = medianOf3(left, right);
         int partition = partitionIt(left, right, median);
         recQuickSort(left, partition-1);
         recQuickSort(partition+1, right);
         }
      }
//-------------------------------------------------------------
   public long medianOf3(int left, int right)
      {
      int center = (left+right)/2;
                                    // Упорядочение left и center
      if( theArray[left] > theArray[center] )
         swap(left, center);
                                    // Упорядочение left и right
      if( theArray[left] > theArray[right] )
         swap(left, right);
                                    // Упорядочение center и right
      if( theArray[center] > theArray[right] )
         swap(center, right);

      swap(center, right-1);        // Размещение медианы на правом краю
      return theArray[right-1];     // Метод возвращает медиану
      }
```

```
//--------------------------------------------------------------
   public void swap(int dex1, int dex2)  // Перестановка двух элементов
      {
      long temp = theArray[dex1];        // A копируется в temp
      theArray[dex1] = theArray[dex2];   // B копируется в A
      theArray[dex2] = temp;             // temp копируется в B
      }
//--------------------------------------------------------------
   public int partitionIt(int left, int right, long pivot)
      {
      int leftPtr = left;             // Справа от первого элемента
      int rightPtr = right - 1;       // Слева от опорного элемента
      while(true)
         {
         while( theArray[++leftPtr] < pivot )  // Поиск большего элемента
            ;                                  // (пустое тело цикла)
         while( theArray[--rightPtr] > pivot ) // Поиск меньшего элемента
            ;                                  // (пустое тело цикла)
         if(leftPtr >= rightPtr)      // Если указатели сошлись,
            break;                    // разбиение закончено
         else                         // В противном случае
            swap(leftPtr, rightPtr);  // Поменять элементы местами
         }
      swap(leftPtr, right-1);         // Восстановление опорного элемента
      return leftPtr;                 // Позиция разбиения
      }
//--------------------------------------------------------------
                                      // Сортировка методом вставки
   public void insertionSort(int left, int right)
      {
      int in, out;
                                      // Сортировка слева от out
      for(out=left+1; out<=right; out++)
         {
         long temp = theArray[out];    // Скопировать помеченный элемент
         in = out;                     // Начать перемещения с out
                                       // Пока не найден меньший элемент
         while(in>left && theArray[in-1] >= temp)
            {
            theArray[in] = theArray[in-1]; // Сдвинуть элемент вправо
            --in;                      // Перейти на одну позицию влево
            }
         theArray[in] = temp;          // Вставить помеченный элемент
         }
      }
//--------------------------------------------------------------
   }  // Конец класса ArrayIns
////////////////////////////////////////////////////////////////
```

продолжение ⇗

Листинг 7.5 (*продолжение*)

```
class QuickSort3App
   {
   public static void main(String[] args)
      {
      int maxSize = 16;             // Размер массива
      ArrayIns arr;                 // Ссылка на массив
      arr = new ArrayIns(maxSize);  // Создание массива

      for(int j=0; j<maxSize; j++)  // Заполнение массива
         {                          // случайными числами
         long n = (int)(java.lang.Math.random()*99);
         arr.insert(n);
         }
      arr.display();                // Вывод элементов
      arr.quickSort();              // Быстрая сортировка
      arr.display();                // Повторный вывод
      }
```

Использование сортировки методом вставки для малых подмассивов в нашей конкретной конфигурации оказалось самым быстрым решением, но оно лишь ненамного быстрее ручной сортировки подмассивов из трех и менее ячеек (как в программе quickSort2.java). Количество сравнений и операций копирования значительно сокращается в фазе сортировки, но при сортировке методом вставок оно возрастает почти в равной степени, так что выигрыш по времени получается незначительным. И все же это решение стоит опробовать, если вы пытаетесь «выжать» из быстрой сортировки всю возможную скорость.

Сортировка методом вставки после быстрой сортировки

Также возможен и другой вариант: быстрая сортировка массива *без* сортировки подмассивов, меньших порога отсечения. Это решение представлено закомментированной строкой в методе quickSort(). (Если вы решите использовать его, исключите вызов insertionSort() из recQuickSort().) После завершения быстрой сортировки массив будет почти упорядочен. После этого сортировка методом вставки применяется ко всему массиву. Этот алгоритм эффективен для почти упорядоченных массивов, поэтому некоторые специалисты рекомендуют это решение, но в нашей конфигурации оно работало очень медленно. Похоже, сортировка методом вставок лучше справляется с множеством мелких сортировок, чем с одной большой.

Устранение рекурсии

Многие авторы также рекомендуют внести еще одно усовершенствование: избавиться от рекурсии в алгоритме быстрой сортировки. Для этого алгоритм переписывается таким образом, чтобы границы подмассива (left и right) сохранялись в стеке, а для разбиения постоянно уменьшающихся подмассивов вместо рекурсии применяется цикл. Все это делается для того, чтобы ускорить работу программы за счет устранения вызовов методов. Однако эта концепция родилась в среде старых компиляторов и компьютерных архитектура, в которых каждый вызов метода

был сопряжен со значительными затратами времени. Трудно сказать, насколько эффективным окажется устранение рекурсии в современных системах, в которых вызовы методов обрабатываются более эффективно.

Эффективность быстрой сортировки

Ранее мы уже упоминали о том, что быстрая сортировка выполняется за время $O(N^*\log N)$. Как было показано при обсуждении сортировки слиянием в главе 6, это характерно для алгоритмов последовательного разделения, в которых рекурсивный метод делит диапазон элементов на две группы, а затем вызывает себя для обработки каждой группы. В этом случае логарифм вычисляется по основанию 2, то есть время выполнения пропорционально $N \times \log_2 N$.

Чтобы получить представление о применимости оценки $N \times \log_2 N$ для быстрой сортировки, запустите одно из приложений QuickSort Workshop для 100 случайных элементов и присмотритесь к пунктирным горизонтальным линиям. Каждая пунктирная линия изображает массив или подмассив, к которому применяется разбиение: указатели `leftScan` и `rightScan` перемещаются навстречу друг другу, сравнивая элементы данных и выполняя перестановку в случае необходимости. В разделе «Разбиение» было показано, что одно разбиение выполняется за время $O(N)$. Таким образом, общая длина всех пунктирных линий пропорциональна времени выполнения быстрой сортировки. Но как ее измерить? Прикладывать линейку к экрану было бы неудобно; к счастью, нужную величину можно вычислить иначе.

Всегда существует одна линия, проходящая по всей ширине диаграммы через N столбцов. Она представляет первое разбиение. Выше и ниже ее располагаются две линии со средней длиной $N/2$; их суммарная длина также равна N. Также имеются 4 линии со средней длиной $N/4$, которые имеют суммарную длину N, затем 8 линий, 16 и т. д. На рис. 7.15 изображена структура разбиения для 1, 2, 4 и 8 линий.

На этом рисунке сплошные горизонтальные линии изображают пунктирные горизонтальные линии из приложений быстрой сортировки, а надписи вида *Длина N/4 ячеек* обозначают среднюю, а не фактическую длину линии. Числа в кружках слева задают порядок создания линий.

Каждая серия линий (например, восемь линий $N/8$) соответствует некоторому уровню рекурсии. Исходный вызов `recQuickSort()` относится к первому уровню и создает первую линию; два вызова из него (второй уровень рекурсии) образуют следующие две линии и т. д. Результаты для 100 ячеек приведены в табл. 7.4.

Когда процесс деления останавливается? Если мы будем делить 100 на 2 и подсчитывать количество делений, то получим ряд 100, 50, 25, 12, 6, 3, 1, соответствующий семи уровням рекурсии. Приложения подтверждают этот результат: если взять точку на диаграмме и подсчитать все пунктирные линии выше и ниже ее, то в среднем их будет семь.

Согласно табл. 7.4 суммарная длина получается равной 652 ячейкам. Из-за ошибок округления оценка получается приближенной, но она близка к логарифму 100 по основанию 2, умноженному на 100, то есть 6,65. Таким образом, неформальный анализ подтверждает правильность оценки времени выполнения $N^*\log_2 N$ для быстрой сортировки.

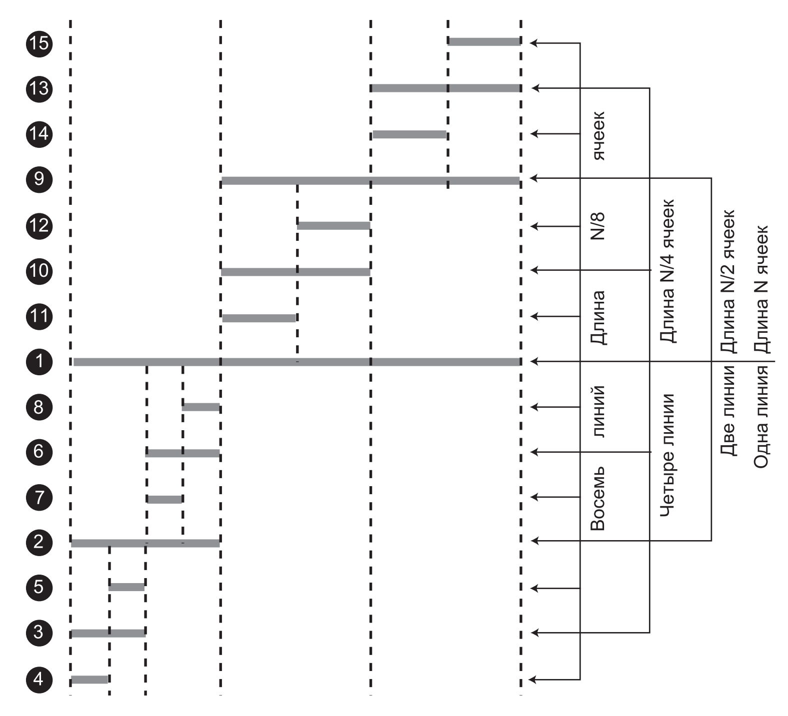


Рис. 7.15. Линии, соответствующие разбиениям

Таблица 7.4. Длины линий и рекурсия

Уровень рекурсии	Номера шагов на рис. 7.15	Средняя длина линии, ячейки	Количество линий	Общая длина, ячейки
1	1	100	1	100
2	2, 9	50	2	100
3	3, 6, 10, 13	25	4	100
4	4, 5, 7, 8, 11, 12, 14, 15	12	8	96
5	Не показано	6	16	96
6	Не показано	3	32	96
7	Не показано	1	64	64
				Итого: 652

Точнее говоря, в разделе, посвященном разбиению, было показано, что разбиение требует $N + 2$ сравнений и менее $N/2$ перестановок. Результаты, полученные при умножении этих величин на $\log_2 N$ для разных значений N, приведены в табл. 7.5.

Величина $\log_2 N$, использованная в табл. 7.5, действует только в лучшем случае, когда каждый подмассив разбивается ровно пополам. Для случайных данных показатели будут чуть больше. Тем не менее приложения QuickSort1 и Quick-

Sort2 Workshop выдают достаточно близкие результаты для 12 и 100 элементов; чтобы убедиться в этом, запустите их и проследите за содержимым полей `Swaps` и `Comparisons`.

Таблица 7.5. Перестановки и сравнения при быстрой сортировке

N	8	12	16	64	100	128
$\log_2 N$	3	3,59	4	6	6,65	7
$N \times \log_2 N$	24	43	64	384	665	896
Сравнения: $(N + 2) \times \log_2 N$	30	50	72	396	678	910
Перестановки: менее $N/2 \times \log_2 N$	12	21	32	192	332	448

Из-за разных порогов отсечения и способов обработки малых подмассивов QuickSort1 Workshop выполняет меньше перестановок, но больше сравнений, чем QuickSort2. В таблице 7.5 указано максимальное количество перестановок (для данных, отсортированных в обратном порядке). Для случайных данных количество перестановок составит от 1/2 до 2/3 приведенных цифр.

Поразрядная сортировка

Глава завершается кратким описанием алгоритма сортировки, основанного на другом принципе. Во всех алгоритмах, приведенных ранее, ключ рассматривался как простое числовое значение, которое сравнивалось с другими значениями для сортировки данных. Поразрядная сортировка разбивает ключ на цифры и упорядочивает элементы в соответствии со значениями цифр. Как ни удивительно, сравнений она вообще не требует.

Алгоритм поразрядной сортировки будет рассматриваться в контексте обычной десятичной системы счисления, так как она более наглядна. Однако эффективная реализация алгоритма использует двоичную систему счисления, чтобы пользоваться преимуществами быстрых компьютерных операций с битами. Мы рассмотрим поразрядную сортировку вместо похожей, но несколько более сложной *обменной поразрядной сортировки*. Сортировка основана на раздельном анализе каждой цифры ключа, начиная с последней (наименее значащей) цифры.

1. Все элементы данных делятся на 10 групп в соответствии со значением последней цифры.
2. Элементы всех 10 групп собираются в единую последовательность: сначала идут все элементы с ключами, завершающимися цифрой 0, затем все элементы с ключами, завершающимися цифрой 1 и так далее до 9. Мы будем называть этот шаг «вспомогательной сортировкой».
3. При второй вспомогательной сортировке все данные снова делятся на 10 групп, но на этот раз в соответствии со значением предпоследней цифры (цифры десятков). Деление не должно нарушать порядка предыдущей сортировки. Иначе

говоря, в каждой из 10 групп порядок элементов остается таким же, как после шага 2 (вспомогательные сортировки должны быть устойчивыми).

4. Все 10 групп снова объединяются. Сначала идут элементы, у которых предпоследняя цифра равна 0, затем элементы с предпоследней цифрой 1 и так далее до 9.
5. Процесс повторяется для всех остальных цифр. Если некоторые ключи содержат меньше цифр, чем другие, считается, что их старшие разряды заполнены 0.

В следующем примере сортируются 7 элементов данных, каждый из которых состоит из трех цифр. Для ясности короткие числа дополнены начальными нулями.

```
421 240 035 532 305 430 124           // Исходный массив
(240 430) (421) (532) (124) (035 305)   // Сортировка по цифре единиц
(305) (421 124) (430 532 035) (240)     // Сортировка по цифре десятков
(035) (124) (240) (305) (421 430) (532) // Сортировка по цифре сотен
035 124 240 305 421 430 532             // Отсортированный массив
```

Круглыми скобками обозначают группы. В каждой группе цифры в соответствующей позиции элементов совпадают. Чтобы убедиться в том, что это решение работает, повторите его вручную для нескольких чисел на листке бумаги.

Проектирование программы

Вероятно, на практике исходные данные будут получены в обычном массиве. Где хранить эти 10 групп? С использованием другого массива или 10 массивов могут возникнуть проблемы — вряд ли количество нулей, единиц, двоек и так далее в каждой позиции будет в точности одинаковым, поэтому размер массива трудно определить заранее. Вместо 10 массивов можно использовать 10 связанных списков, способных сокращаться и расширяться по мере надобности. Именно так мы и поступим.

Внешний цикл последовательно перебирает все цифры ключа. В нем выполняются два внутренних цикла: первый читает данные из массива и размещает их в списках, а второй копирует из списков обратно в массив. Обратите внимание на выбор правильной разновидности связанного списка. Чтобы вспомогательная сортировка была устойчивой, данные должны читаться из списков в порядке их сохранения. Какой тип связанных списков позволяет легко решить эту задачу? Подробности реализации остаются читателю для самостоятельной работы.

Эффективность поразрядной сортировки

На первый взгляд эффективность поразрядной сортировки выглядит неправдоподобно хорошо. От вас требуется лишь копировать данные из массивов в списки и обратно. Для 10 элементов данных будет выполнено 20 операций копирования. Процедура повторяется по одному разу для каждой цифры. Если, скажем, число состоит из 5 цифр, выходит $20 \times 5 = 100$ операций копирования. Для 100 элементов данных получаем $200 \times 5 = 1000$ операций копирования. Количество копирований пропорционально количеству элементов, то есть сортировка обладает сложностью $O(N)$ — самый эффективный алгоритм сортировки из всех, которые нам встречались.

К сожалению, при увеличении количества элементов обычно требуются более длинные ключи. Если объем данных увеличивается в 10 раз, ключ обычно приходится дополнять новой цифрой. Таким образом, количество операций копирования пропорционально количеству элементов, умноженному на количество цифр в ключе. Последнее вычисляется как логарифм от значений ключа, так что в большинстве случаев мы возвращаемся к эффективности $O(N \times \log N)$ — такой же, как у быстрой сортировки.

Сравнения при поразрядной сортировке не выполняются, хотя на извлечение каждой цифры из числа также требуется время. Операция должна выполняться по одному разу для каждых двух копирований. Впрочем, на конкретном компьютере извлечение цифр в двоичном виде может выполняться быстрее, чем сравнение. Конечно, поразрядная сортировка (как и сортировка слиянием) использует приблизительно вдвое больше памяти, чем быстрая сортировка.

Итоги

- Сортировка Шелла применяет сортировку методом вставки к элементам, разделенных большими интервалами; затем интервалы сокращаются и т. д.
- Выражением «n-сортировка» обозначается сортировка каждого n-го элемента.
- Для определения интервалов сортировки в алгоритме Шелла используется последовательность чисел, называемая *интервальной последовательностью*.
- Часто применяемая интервальная последовательность генерируется по рекурсивной формуле $h = 3 \times h + 1$, где исходное значение h равно 1.
- С массивом из 1000 элементов будут последовательно выполнены 364-сортировка, 121-сортировка, 40-сортировка, 13-сортировка, 4-сортировка и, наконец, 1-сортировка.
- Эффективность алгоритма Шелла трудно оценить теоретически, но сортировка выполняется приблизительно за время $O(N \times (\log N))$. Это намного быстрее, чем у алгоритмов с временем выполнения $O(N2)$ (таких, как сортировка методом вставки), но медленнее, чем у алгоритмов $O(N \times \log N)$ — например, быстрой сортировки.
- *Разбиением* массива называется его деление на два подмассива, один из которых содержит элементы с ключом ниже заданной величины, а другой — элементы с ключом большим либо равным этой величины.
- *Опорным значением* называется значение, определяющее принадлежность элемента к той или иной группе в процессе разбиения. Элементы, меньшие опорного значения, попадают в левую группу; бóльшие элементы попадают в правую группу.
- В алгоритме разбиения два индекса (каждый в своем цикле `while`) начинают двигаться навстречу друг другу с противоположных концов массива. В процессе перемещения они ищут элементы, которые необходимо переставить.

- Когда индекс находит элемент для перестановки, его цикл `while` завершается.
- При выходе из обоих циклов `while` элементы меняются местами.
- Когда происходит выход из обоих циклов `while`, а индексы встретились или прошли мимо друг друга, разбиение завершено.
- Разбиение выполняется за линейное время $O(N)$, с выполнением $N + 1$ или $N + 2$ сравнений и менее чем $N/2$ перестановок.
- Алгоритм разбиения может потребовать дополнительных проверок во внутренних циклах `while` для предотвращения выхода индексов за границы массива.
- Метод быстрой сортировки выполняет разбиение массива, а затем дважды рекурсивно вызывает себя для сортировки двух полученных подмассивов.
- Подмассив из одного элемента уже отсортирован; он может стать базовым ограничением для быстрой сортировки.
- Опорным значением для разбиения при быстрой сортировке является значение ключа конкретного элемента, называемого *опорным элементом*.
- В простой версии быстрой сортировки в качестве опорного элемента можно использовать элемент у правого края подмассива.
- В ходе разбиения опорный элемент не включается в перебор и не участвует в процессе разбиения.
- Позднее опорный элемент копируется в ячейку между двумя подмассивами. Там он занимает свою окончательную позицию в порядке сортировки.
- В простой версии быстрая сортировка уже упорядоченных (в прямом или обратном порядке) данных выполняется за время $O(N^2)$.
- В более сложной версии быстрой сортировки опорный элемент может определяться как медиана первого, последнего и среднего элементов подмассива. Этот прием называется *определением медианы по трем точкам*.
- Определение медианы по трем точкам фактически решает проблему $O(N^2)$ для предварительно отсортированных данных.
- При определении медианы по трем точкам левый, средний и правый элементы сортируются одновременно с определением медианы.
- Этот вариант сортировки отменяет необходимость проверки конца массива во внутренних циклах `while` в алгоритмах разбиения.
- Быстрая сортировка выполняется за время $O(N \times \log_2 N)$ (не считая применения простой версии к упорядоченным данным).
- Подмассивы, размер которых меньше некоторого порога (порог отсечения), могут сортироваться другим алгоритмом, отличным от быстрой сортировки.
- Для сортировки подмассивов, размер которых меньше порога отсечения, обычно применяется сортировка методом вставки.
- Сортировка методом вставки также может быть применена ко всему массиву — после того, как он был отсортирован до порога отсечения посредством быстрой сортировки.
- Поразрядная сортировка работает почти так же быстро, как быстрая сортировка, но расходует вдвое больше памяти.

Вопросы

Следующие вопросы помогут читателю проверить качество усвоения материала. Ответы на них приведены в приложении В.

1. Сортировка Шелла основана на:
   a) разбиении массива;
   b) перестановке соседних элементов;
   c) решении проблемы элементов, находящихся на больших расстояниях;
   d) предварительном выполнении обычной сортировки методом вставки.
2. Если массив содержит 100 элементов, то алгоритм Кнута начнет с интервала _________.
3. Какие из следующих действий *не следует* выполнять для преобразования сортировки методом вставки в сортировку Шелла?
   a) Замена 1 на h.
   b) Вставка алгоритма создания интервалов уменьшающейся ширины.
   c) Заключение обычной сортировки методом вставки в цикл.
   d) Изменение направления смещения индексов во внутреннем цикле.
4. Хорошая интервальная последовательность сортировки Шелла строится многократным делением размера массива на 2 (Да/Нет).
5. Заполните пропуски в O-синтаксисе: скорость сортировки Шелла выше ________, но ниже ________ .
6. Разбиением называется:
   a) группировка всех элементов, бо́льших некоторого значения, у одного конца массива;
   b) деление массива пополам;
   c) частичная сортировка частей массива;
   d) раздельная сортировка каждой половины массива.
7. При разбиении каждый элемент массива сравнивается с ________.
8. Если при разбиении элемент массива равен ответу к вопросу 7, то он:
   a) пропускается;
   b) пропускается или нет в зависимости от другого элемента массива;
   c) помещается в позицию опорного элемента;
   d) меняется с ним местами.
9. При быстрой сортировке в качестве опорного элемента может выбираться любой элемент массива (Да/Нет)
10. Если большие ключи сгруппированы справа, то индексом разбиения называется:
    a) индекс элемента между левым и правым подмассивами;
    b) значение ключа элемента между левым и правым подмассивами;

c) индекс левого элемента правого подмассива;

d) значение ключа левого элемента правого подмассива.

11. Быстрая сортировка основана на разбиении исходного массива с последующим __________.
12. После разбиения в простой версии быстрой сортировки опорный элемент:

a) используется для определения медианы массива;

b) меняется местами с элементом правого подмассива;

c) используется в качестве отправной точки следующего разбиения;

d) уничтожается.

13. Определение медианы по трем точкам используется для выбора ________ .
14. При быстрой сортировке массива из N элементов метод `partitionIt()` проверяет каждый элемент приблизительно _______ раз.
15. Быструю сортировку можно ускорить, прерывая разбиение, когда размер подмассива уменьшится до 5, и завершая его с другим алгоритмом сортировки (Да/Нет).

Упражнения

Упражнения помогут вам глубже усвоить материал этой главы. Программирования они не требуют.

1. Проверьте, что произойдет при выполнении приложения Partition Workshop для 100 элементов, отсортированных в обратном порядке.
2. Измените программу `shellSort.java` (листинг 7.1) так, чтобы после завершения каждой *n*-сортировки выводилось все содержимое массива. Массив должен быть достаточно маленьким, чтобы его содержимое помещалось в одной строке. Проанализируйте промежуточный вывод и убедитесь в том, что алгоритм работает так, как предполагалось.
3. Измените программы `shellSort.java` (листинг 7.1) и `quickSort3.java` (листинг 7.5) так, чтобы они сортировали массивы достаточно большого размера, и сравните их по скорости. Также сравните скорость работы этих алгоритмов со скоростью работы алгоритмов из главы 3.

Программные проекты

В этом разделе предлагаются программные проекты, которые укрепляют понимание материала и демонстрируют практическое применение концепций этой главы.

7.1. Измените программу `partition.java` (листинг 7.2) так, чтобы метод `partitionIt()` всегда использовал в качестве опорного элемент с наибольшим индексом (крайний правый) вместо произвольного числа. (По аналогии с программой `quickSort1.java` в листинге 7.3.) Убедитесь в том, что метод будет работать для массивов из трех и менее элементов. Для этого придется включить в программу несколько дополнительных команд.

7.2. Измените программу `quickSort2.java` (листинг 7.4) так, чтобы она подсчитывала количество копирований и сравнений во время сортировки, а затем выводила результаты. Программа должна обеспечивать ту же эффективность, что и приложение QuickSort2 Workshop, поэтому количество операций копирования и сравнения у них должно совпадать. (Не забудьте, что перестановка состоит из трех операций копирования.)

7.3. В упражнении 3.2 главы 3 для поиска медианы набора данных предлагалось отсортировать данные и выбрать средний элемент. Казалось бы, быстрая сортировка с выбором среднего элемента является самым быстрым способом определения медианы, однако в действительности существует еще более быстрое решение, которое использует алгоритм разбиения для определения медианы без полной сортировки данных.

Представьте, что после разбиения данных опорный элемент по случайности совпал со средним элементом. Готово! Все элементы справа от опорного элемента больше (либо равны), все элементы слева меньше (либо равны); значит, если опорный элемент приходится на точный центр массива, это и есть медиана. Конечно, такое совпадение встречается не так часто, но положение можно исправить повторным разбиением подмассива, содержащего средний элемент.

Допустим, массив состоит из семи элементов с номерами от 0 до 6. Средним является элемент 3. Если после разбиения массива опорным элементом окажется 4, нужно снова провести разбиение от 0 до 4 (подмассив, содержащий 3), не включая элементы 5 и 6. Если опорным элементом окажется 2, то разбиение проводится в подмассиве от 2 до 6, не включая 0 и 1. Далее вы продолжаете рекурсивное разбиение соответствующих подмассивов, постоянно проверяя, не совпадает ли опорный элемент со средним элементом. В конечном итоге это произойдет — задача решена. Так как для этого потребуется меньше разбиений, чем при быстрой сортировке, алгоритм работает быстрее.

Расширьте программный проект 7.1 и реализуйте определение медианы массива. В нем будут использоваться рекурсивные вызовы, аналогичные тем, которые используются при быстрой сортировке, но они будут только разбивать подмассивы без их полной сортировки. Процесс прерывается при обнаружении медианы, а не при завершении сортировки массива.

7.4. *Выбором* называется поиск k-го большего или меньшего элемента в массиве (например, выбор 7-го по величине элемента). Определение медианы (см. программный проект 7.2) является особым случаем выбора. Для выбора можно использовать тот же процесс разбиения, но вместо среднего элемента в этом случае ищется элемент с заданным индексом. Измените программу из проекта 7.2 так, чтобы она поддерживала выбор произвольного элемента. С каким минимальным размером массива будет работать ваша программа?

7.5. Реализуйте поразрядную сортировку, описанную в последнем разделе этой главы. Она должна поддерживать переменные объемы данных и количество цифр в ключе. Одно из возможных расширений — введение отдельной переменной для основания системы счисления, чтобы она могла быть отличной от 10, но в этом случае вам будет трудно разобраться в результатах (проблема может быть решена написанием метода вывода числовых данных в разных системах счисления).

Глава 8

Двоичные деревья

От алгоритмов, которые были основной темой главы 7, «Нетривиальная сортировка», в этой главе мы переходим к структурам данных. Двоичные деревья принадлежат к числу основных структур данных, используемых в программировании. Они обладают некоторыми уникальными преимуществами, которых не было у других структур данных, рассматривавшихся нами ранее. В этой главе вы узнаете, для чего используются деревья, как они работают и как их лучше создавать.

Для чего нужны двоичные деревья?

Почему программисты используют двоичные деревья в своих программах? Обычно потому, что двоичное дерево сочетает в себе преимущества двух других структур: упорядоченного массива и связанного списка. Поиск в дереве выполняется так же быстро, как в упорядоченном массиве, а операции вставки и удаления элементов так же быстро, как в связанном списке. Давайте немного подробнее рассмотрим эти темы, прежде чем переходить к подробностям реализации деревьев.

Медленная вставка в упорядоченном массиве

Представьте массив, все элементы которого хранятся в порядке сортировки, то есть упорядоченный массив (см. главу 2, «Массивы»). Как вы уже знаете, такой массив позволяет быстро найти конкретный элемент с конкретным значением ключа посредством двоичного поиска. Алгоритм проверяет середину массива; если искомый ключ больше найденного, поиск ограничивается верхней половиной массива, а если меньше — нижней половиной. Многократное применение этого процесса обеспечивает нахождение объекта за время $O(\log N)$. Упорядоченный массив также поддерживает быстрый перебор с посещением объектов в порядке сортировки.

С другой стороны, чтобы вставить новый объект в упорядоченный массив, необходимо сначала определить позицию для вставки, а затем переместить все объекты с бóльшим значением ключа к концу массива, чтобы освободить место для нового элемента. Все эти перемещения занимают много времени, так как в среднем сдвигается половина элементов ($N/2$). Удаление тоже требует множественных смещений, а следовательно, тоже является медленной операцией.

Если вы собираетесь часто выполнять операции вставки и удаления, упорядоченный массив будет плохим вариантом.

Медленный поиск в связанном списке

С другой стороны, как было показано в главе 5, «Связанные списки», операции вставки и удаления в связанных списках выполняются относительно быстро — в структуре данных достаточно изменить лишь несколько ссылок. Иначе говоря, операции выполняются за время $O(1)$ (наибольшая эффективность в O-записи).

К сожалению, с *поиском* элемента в связанном списка дело обстоит сложнее. Алгоритм вынужден последовательно перебирать элементы от начала списка, пока не будет найден искомый элемент. Таким образом, в переборе будет задействовано в среднем $N/2$ элементов, а ключ каждого элемента будет сравниваться с искомым значением. Операция выполняется за время $O(N)$, что считается медленным. (Обратите внимание: время выполнения, которое считается быстрым для сортировки, при операциях со структурами данных оказывается медленным.)

Казалось бы, для ускорения операций можно воспользоваться упорядоченным связанным списком, в котором все элементы расположены в порядке сортировки, но эта мера не помогает. Алгоритму все равно приходится последовательно перебирать элементы от начала, так как к элементу невозможно обратиться без отслеживания всей предшествующей цепочки ссылок. (Конечно, в упорядоченном списке перебор узлов по порядку выполняется намного быстрее, чем в неупорядоченном, но это не помогает найти произвольный объект.)

Деревья приходят на помощь

В идеале нам хотелось бы иметь структуру данных с быстрыми операциями вставки/удаления, как в связанном списке, и быстрым поиском, как в упорядоченном массиве. Деревья обладают обеими характеристиками; кроме того, это одна из самых интересных структур данных.

Что называется деревом?

Нас прежде всего интересует конкретная разновидность деревьев, называемая *двоичным деревом*. Но прежде чем браться за эту тему, необходимо разобраться в том, что же называется деревом.

Дерево состоит из *узлов*, соединенных *ребрами*. На рис. 8.1 изображен пример дерева. В таком представлении дерева (а также в нашем приложении Workshop) узлы обозначаются кружками, а ребра — линиями, соединяющими кружки.

Деревья как абстрактная математическая категория были достаточно подробно изучены. В действительности дерево является частным случаем более общей структуры, называемой *графом*, но для нас это пока несущественно. Графы будут рассматриваться в главе 13, «Графы», и главе 14, «Взвешенные графы».

В программах узлы часто представляют сущности: людей, детали машин, забронированные авиабилеты и т. д., то есть типичные элементы, сохраняемые в любых структурах данных. В ООП-языках, к числу которых относится Java, сущности реального мира представляются в виде объектов.

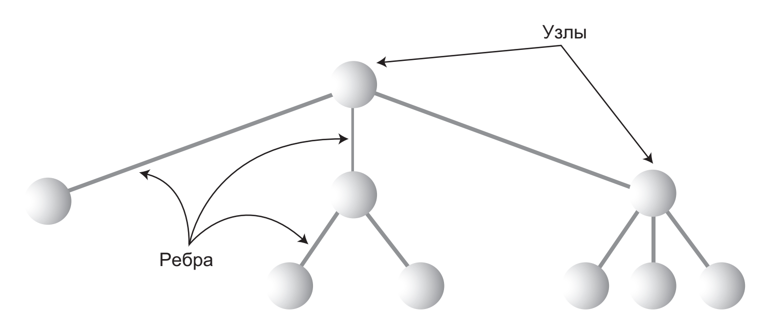


Рис. 8.1. Обобщенное (не двоичное) дерево

Ребра (соединительные линии между узлами) представляют отношения между узлами. Упрощенно говоря, программа может легко (и быстро) перейти от узла к узлу, если между ними имеется соединительная линия. Более того, переходы между узлами возможны *только* по соединительным линиям. В общем случае перемещение происходит только в одном направлении: от корневого узла вниз.

В программах, написанных на Java, ребра обычно представляются ссылками (а в программах на C или C++ используются указатели).

Обычно на верхнем уровне дерева располагается один узел, который соединяется с другими узлами на втором уровне; те, в свою очередь, соединяются с еще большим количеством узлов на третьем уровне и т. д. Таким образом, деревья постепенно расширяются сверху вниз. По сравнению с настоящим деревом они перевернуты, но обычно программа начинает обработку с корневого узла, а операции (по распространенному мнению) более естественно представляются в нисходящем виде, как при чтении текста.

Существует несколько разновидностей деревьев. У дерева на рис. 8.1 узлы имеют более двух потомков (см. ниже). Однако эта глава посвящена особой разновидности деревьев — так называемым двоичным деревьям. Каждый узел двоичного дерева имеет не более двух потомков. Более общие деревья, в которых узлы могут иметь более двух потомков, называются *многопутевыми деревьями*. Пример будет представлен в главе 10, «Деревья 2-3-4».

Терминология

Для описания различных аспектов деревьев применяются разные термины. Чтобы наше обсуждение было содержательным, необходимо знать хотя бы часть из них. К счастью, многие термины имеют отношение к «настоящим» деревьям или семейным отношениям (родители/потомки), так что запомнить их несложно. На рис. 8.2 показаны некоторые термины применительно к двоичному дереву.

Путь

Представьте, как кто-то переходит от узла к узлу по соединяющим их ребрам. Полученная последовательность узлов называется *путем* (path).

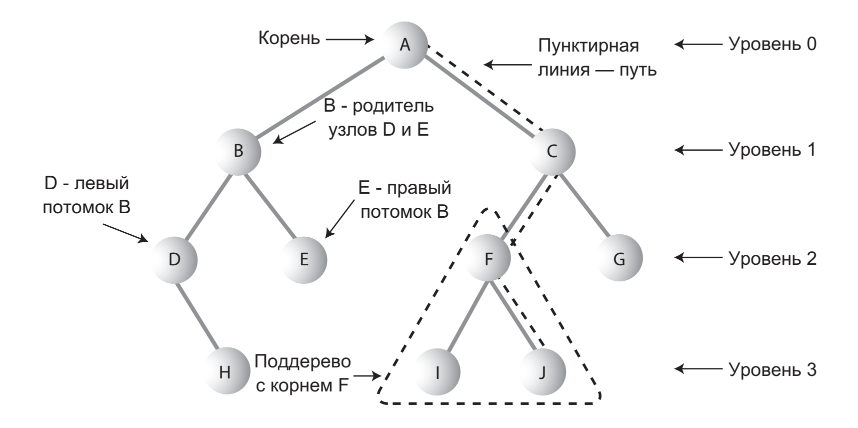


Рис. 8.2. Термины, используемые при работе с деревьями

Корень

Узел на верхнем уровне дерева называется *корневым узлом* (*корнем*). Дерево имеет только один корень. Чтобы совокупность узлов и ребер могла называться деревом, от корня к любому другому узлу должен вести один (и только один!) путь. Структура на рис. 8.3, деревом не является, поскольку нарушает это правило.

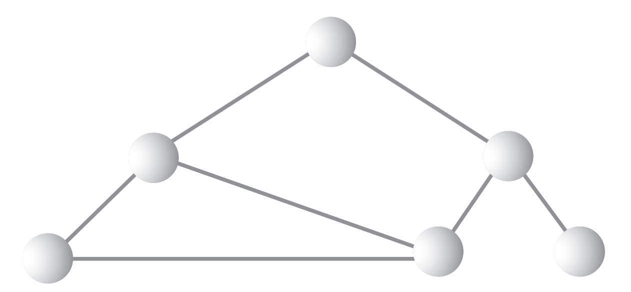


Рис. 8.3. Структура, не являющаяся деревом

Родитель

Любой узел (кроме корневого) имеет ровно одно ребро, уходящее вверх к другому узлу. Узел, расположенный выше него, называется *родительским узлом* (или просто *родителем*) по отношению к данному узлу.

Потомок

Любой узел может иметь одно или несколько ребер, соединяющих его с узлами более низкого уровня. Такие узлы, находящиеся ниже заданного узла, называются его *потомками*.

Лист

Узел, не имеющий потомков, называется *листовым узлом* (или просто *листом*). Дерево всегда имеет только один корень, но листьев может быть несколько.

Поддерево

Любой узел может рассматриваться как корень поддерева, состоящего из его потомков, потомков его потомков и т. д.

Посещение

Переход программы к узлу (обычно с целью выполнения некоторой операции, например проверки значения одного из полей данных или вывода) называется *посещением*. Простое прохождение мимо узла на пути от одного узла к другому посещением не считается.

Обход

Обходом дерева называется посещение всех его узлов в некотором заданном порядке. Например, все узлы дерева могут перебираться в порядке возрастания ключей. Как будет вскоре показано, существуют и другие способы обхода деревьев.

Уровни

Уровенем узла называется количество поколений, отделяющих его от корня. Если считать, что корень находится на уровне 0, то его потомки находятся на уровне 1, потомки потомков — на уровне 2 и т. д.

Ключи

Как вы уже знаете, одно из полей данных объекта часто назначается *ключевым*. Ключ используется при поиске элемента или выполнения с ним других операций. На древовидных диаграммах узел, содержащий данные, обычно обозначается кружком, а внутри кружка отображается значение ключа. (Это обозначение еще встретится вам на многих диаграммах.)

Двоичное дерево

Если каждый узел дерева имеет не более двух потомков, такое дерево называется *двоичным*. В этой главе мы будем рассматривать двоичные деревья, потому что это самая простая и самая распространенная разновидность.

Два потомка каждого узла двоичного дерева называются *левым потомком* и *правым потомком* в зависимости от позиции на изображении дерева (рис. 8.2). Число потомков узла двоичного дерева не обязано быть равным 2; узел может иметь только левого или только правого потомка или не иметь потомков вообще (листовой узел).

Формально двоичные деревья, рассматриваемые в этой главе, называются *деревьями двоичного поиска*. Пример дерева двоичного поиска представлен на рис. 8.4.

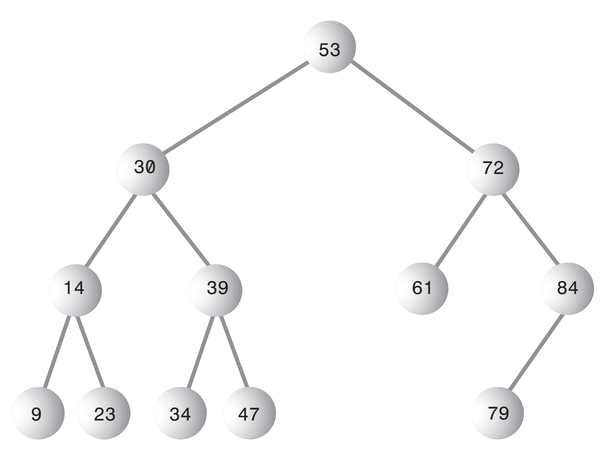


Рис. 8.4. Дерево двоичного поиска

ПРИМЕЧАНИЕ

Определяющая характеристика дерева двоичного поиска: ключ левого потомка узла должен быть меньше, чем у родителя, а ключ правого потомка — больше либо равен ключу родителя.

Аналогия

Распространенный пример дерева — иерархическая файловая структура в компьютерной системе. Корневой каталог устройства (во многих системах обозначаемый символом `\` — как в `C:\`) является корнем дерева. Каталоги, непосредственно вложенные в корневой каталог, являются его потомками. Количество уровней вложения подкаталогов может быть сколь угодно большим. Файлы соответствуют листьям, так как они не имеют потомков.

Конечно, иерархическая файловая структура не является двоичным деревом, потому что каталог может иметь много потомков. Полное имя файла (например, `C:\SALES\EAST\NOVEMBER\SMITH.DAT`) соответствует пути от корня к листу `SMITH.DAT`. Термины описания файловых структур (корень и путь) были позаимствованы из теории деревьев.

Иерархическая файловая структура принципиально отличается от деревьев, которые мы будем рассматривать. В файловой структуре подкаталоги не содержат данных; в них хранятся только ссылки на другие подкаталоги или файлы. Данные хранятся только в файлах. В дереве каждый узел содержит данные (сведения о работнике, спецификации автомобильных деталей и т. д.). Помимо данных, все узлы, кроме листовых, содержат ссылки на другие узлы.

Как работают двоичные деревья?

В этом разделе описано выполнение основных операций с двоичными деревьями: поиск узла с заданным ключом, вставка нового узла, обход дерева и удаление узла. Для каждой из этих операций вы сначала увидите, как соответствующая операция выполняется в приложении Binary Tree Workshop; после этого мы проанализируем соответствующий код Java.

Приложение Binary Tree Workshop

Запустите приложение Binary Tree Workshop. Примерный вид приложения показан на рис. 8.5. (Дерево генерируется случайным образом, поэтому полное совпадение маловероятно.)

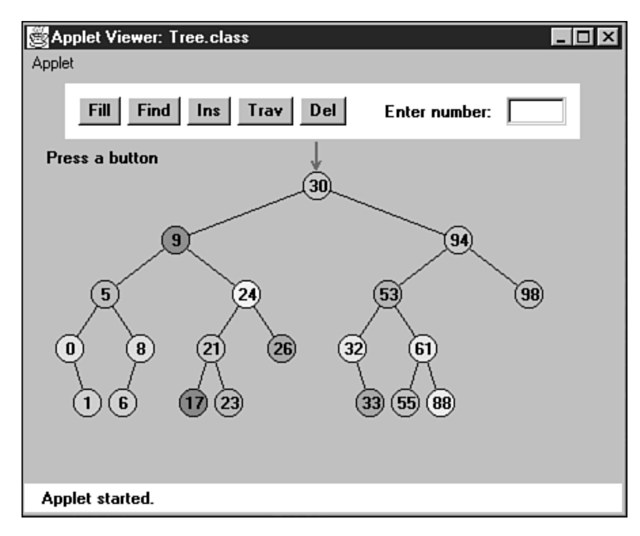


Рис. 8.5. Приложение Binary Tree Workshop

Работа с приложением

Значения ключей, отображаемые в узлах, лежат в диапазоне от 0 до 99. Конечно, в настоящем дереве диапазон значений ключей будет намного шире. Например, если в качестве ключей используются номера социального страхования работников, то верхняя граница диапазона расширяется до 999 999 999.

Другое отличие приложения Workshop от реальных деревьев заключается в том, что приложение Workshop ограничивается максимальной глубиной в 5 уровней; другими словами, от корня до низа дерева не может проходить более 5 уровней. Такое ограничение гарантирует, что все узлы дерева будут видны в приложении. В реальных деревьях количество уровней не ограничивается (новые узлы могут создаваться вплоть до исчерпания свободной памяти в системе).

Чтобы создать новое дерево в приложении Workshop, щелкните на кнопке `Fill`. Вам будет предложено ввести количество узлов в дереве. Величина может изменяться в диапазоне от 1 до 31; значение 15 дает достаточно типичное дерево. После ввода числа дважды нажмите `Fill`, чтобы сгенерировать новое дерево. Приложение дает возможность поэкспериментировать с разным количеством узлов.

Несбалансированные деревья

Некоторые деревья, генерируемые приложением, являются *несбалансированными*, то есть большинство узлов сосредоточено с одной или с другой стороны корня (рис. 8.6). Несбалансированными также могут быть отдельные поддеревья.

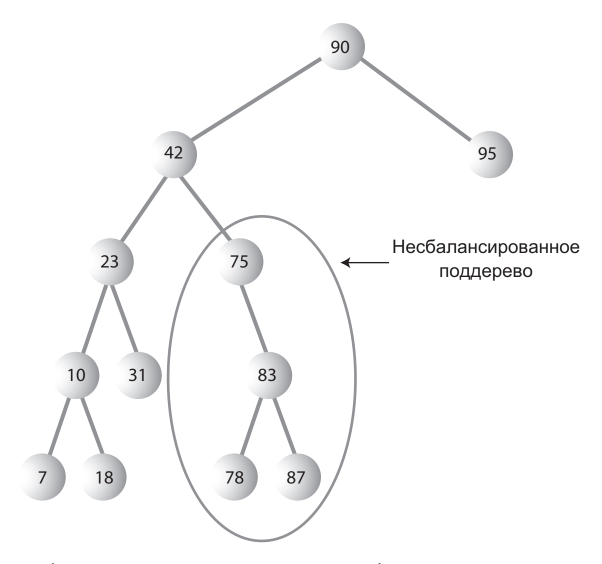


Рис. 8.6. Несбалансированное дерево (с несбалансированным поддеревом)

Деревья становятся несбалансированными из-за порядка вставки элементов данных. Если бы ключи вставлялись случайным образом, то и дерево было бы более или менее сбалансированным. Однако при генерировании восходящей последовательности (11, 18, 33, 42, 65 и т. д.) или нисходящей последовательности все значения будут соответственно правыми или левыми потомками, а дерево станет несбалансированным. Значения ключей в приложении генерируются случайным образом, но, конечно, это приводит к генерированию коротких восходящих или нисходящих последовательностей — и как следствие, локальной несбалансированности. Научившись вставлять элементы в дерево в приложении Workshop, вы сможете построить дерево посредством вставки таких упорядоченных последовательностей — посмотрите, что при этом произойдет.

Если указать достаточно большое количество узлов при создании дерева кнопкой `Fill`, иногда создаваемое дерево не содержит все запрашиваемые узлы. В зависимости от несбалансированности дерева некоторые ветви не могут содержать

полное количество узлов. Это связано с тем, что глубина дерева в приложении ограничена пятью уровнями; в реальных программах таких проблем не бывает.

Если дерево создается на базе элементов данных, ключи которых поступают в случайном порядке, проблема несбалансированности дерева не создает особых проблем, потому что вероятность генерирования длинных серий невелика. Однако значения ключей могут поступать в строгой последовательности — например, если оператор ввода данных перед началом работы раскладывает стопку личных дел по возрастанию табельного номера. В таких ситуациях эффективность дерева может значительно снижаться. Проблемы несбалансированных деревьев и возможные пути решения рассматриваются в главе 9, «Красно-черные деревья».

Представление деревьев в коде Java

Давайте посмотрим, как двоичные деревья реализуются в Java. Как и в случае с другими структурами данных, возможны разные способы представления дерева в памяти компьютера. В самом распространенном представлении узлы хранятся в несмежных блоках памяти, а для их связывания в каждый узел включаются ссылки на его потомков.

Дерево в памяти также может быть представлено в виде массива; узлы, находящиеся в конкретных позициях, хранятся в соответствующих позициях массива. Мы вернемся к этому способу хранения в конце настоящей главы. В следующих примерах кода Java используется модель с соединением узлов посредством ссылок.

ПРИМЕЧАНИЕ

При описании отдельных операций будут приводиться фрагменты кода, относящиеся к данной конкретной операции. Полная программа, из которой были позаимствованы эти фрагменты, приведена в конце главы (листинг 8.1).

Класс Node

Для начала нам понадобится класс для представления объектов узлов. Класс содержит данные, представляющие хранимые объекты (например, описания работников для базы данных отдела кадров), а также ссылки на каждого из двух потомков текущего узла. Определение класса выглядит так:

```
class Node
   {
   int iData;                  // Данные, используемые в качестве ключа
   double fData;               // Другие данные
   node leftChild;             // Левый потомок узла
   node rightChild;            // Правый потомок узла

   public void displayNode()
      {
      // (Тело метода см. в листинге 8.1)
      }
   }
```

Некоторые программисты также включают ссылку на родительский узел. Наличие таких ссылок упрощает одни операции, но усложняет другие, поэтому мы их не используем. В класс также включен метод displayNode() для вывода данных узла, но его код для нас пока не актуален.

Существуют и другие подходы к проектированию класса Node. Вместо размещения данных непосредственно в узле можно воспользоваться ссылкой на объект, представляющий набор данных:

```
class Node
   {
   person p1;                    // Ссылка на объект person
   node leftChild;               // Левый потомок узла
   node rightChild;              // Правый потомок узла
   }

class person
   {
   int iData;
   double fData;
   }
```

Такая архитектура на концептуальном уровне более четко показывает, что узел и хранящиеся в нем данные — не одно и то же, однако она несколько усложняет программный код, поэтому мы будем придерживаться первого решения.

Класс Tree

Нам также понадобится класс для представления не отдельных узлов, а всего дерева. Этот класс будет называться Tree. Он содержит только одно поле: переменную Node, в которой хранится корень дерева. Поля для других узлов не нужны, поскольку доступ к ним осуществляется через корневой узел.

Класс Tree содержит ряд методов для поиска, вставки и удаления узлов, различных видов обхода и вывода содержимого дерева. «Скелет» класса выглядит так:

```
class Tree
   {
   private Node root;           // Единственное поле данных

   public void find(int key)
      {
      }
   public void insert(int id, double dd)
      {
      }
   public void delete(int id)
      {
      }
   // Другие методы
}  // Конец класса Tree
```

Класс TreeApp

Наконец, с созданным деревом необходимо выполнять операции. Следующий класс с методом main() создает дерево, вставляет в него три узла, а затем выполняет поиск одного из них. Назовем этот класс TreeApp:

```
class TreeApp
   {
   public static void main(String[] args)
      {
      Tree theTree = new Tree;        // Создание дерева
      theTree.insert(50, 1.5);        // Вставка трех узлов
      theTree.insert(25, 1.7);
      theTree.insert(75, 1.9);

      node found = theTree.find(25);  // Поиск узла с ключом 25
      if(found != null)
         System.out.println("Found the node with key 25");
      else
         System.out.println("Could not find node with key 25");
      }
   }  // Конец класса TreeApp
```

ПОДСКАЗКА

В листинге 8.1 метод main() также предоставляет примитивный интерфейс, при помощи которого пользователь может выбрать с клавиатуры выполняемую операцию (вставка, поиск, удаление и т. д.).

В следующем разделе рассматриваются отдельные операции с деревьями: поиск, вставка, перебор и удаление узлов.

Поиск узла

Поиск узла с заданным ключом — простейшая из основных операций с деревьями, поэтому мы начнем с нее.

Как уже говорилось ранее, узлы в дереве двоичного поиска соответствуют объектам, содержащим информацию. Это могут быть объекты person с ключевым полем табельного номера, а также полями имени, адреса, телефона, зарплаты и т. д., или объекты, представляющие детали машин, с ключевым полем кода детали и полями текущего количества, цены и т. д. Впрочем, в приложении Workshop для каждого узла хранятся только две характеристики: цвет и число. Узел создается с этими характеристиками и сохраняет их на всем протяжении своего существования.

Поиск узла в приложении Workshop

Выберите в приложении Workshop узел, желательно расположенный в нижней части дерева (как можно дальше от корня). Число, отображаемое в этом узле,

является значением его ключа. Сейчас мы покажем, как приложение Workshop ищет узел по известному ключу.

Для определенности будем считать, что вам потребовалось найти узел, представляющий элемент с ключом 57 (рис. 8.7). Конечно, при запуске приложения Workshop вы получите другое дерево и вам придется выбрать другой ключ.

Щелкните на кнопке `Find`. Приложение запрашивает ключевое значение искомого узла. Введите 57 (или другое число на выбранном вами узле). Щелкните на кнопке `Find` еще два раза.

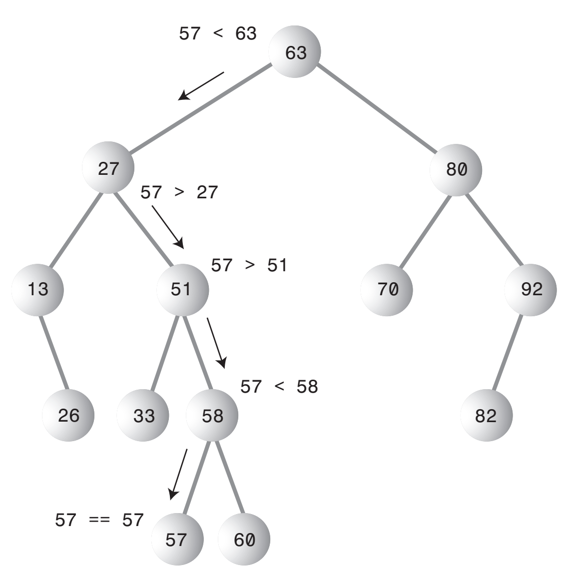


Рис. 8.7. Поиск узла 57

В ходе поиска в приложении выводится сообщение `Going to left child` или `Going to right child`, а красная стрелка смещается на один уровень вниз — соответственно к левому или правому потомку.

На рис. 8.7 перемещение стрелки начинается от корневого узла. Программа сравнивает ключ 57 с ключом корневого узла, равным 63. Искомый ключ меньше, поэтому программа знает, что искомый узел находится в левой части дерева — либо это левый потомок корневого узла, либо один из его потомков. Ключ левого потомка корневого узла равен 27; сравнение 57 и 27 показывает, что искомый узел принадлежит правому поддереву узла 27. Стрелка переходит к узлу 51 — корню этого поддерева. Значение 57 больше 51, поэтому процесс поиска сначала переходит направо к 58, а затем налево к 57. На этот раз ключ 57 совпадает с искомым значением — узел успешно найден.

Приложение Workshop ничего не делает с найденным узлом, только выводит сообщение об успешном поиске. В реальных программах с найденным узлом обычно выполняется какая-нибудь операция — вывод его содержимого, изменение одного из полей и т. д.

Реализация поиска узла на языке Java

Код функции поиска find(), оформленной в виде метода класса Tree:

```
public Node find(int key)     // Поиск узла с заданным ключом
   {                          // (предполагается, что дерево не пустое)
   Node current = root;              // Начать с корневого узла
   while(current.iData != key)       // Пока не найдено совпадение
      {
      if(key < current.iData)        // Двигаться налево?
         current = current.leftChild;
      else
         current = current.rightChild; // Или направо?
      if(current == null)            // Если потомка нет,
         return null;                // поиск завершился неудачей
      }
   return current;                   // Элемент найден
   }
```

Для хранения узла, проверяемого в настоящий момент, используется переменная current. Искомое значение хранится в аргументе key. Поиск начинается с корневого узла (это необходимо, так как в дереве напрямую доступен только этот узел), то есть в начале работы current присваивается ссылка на корневой узел.

Затем в цикле while искомое значение key сравнивается со значением поля iData (ключевого поля) текущего узла. Если значение key меньше, то current присваивается ссылка на левого потомка, а если больше (или равно) — то ссылка на правого потомка узла.

Неудачный поиск

Если ссылка current становится равной null, значит, найти следующего потомка не удалось; перебор достиг конца дерева, искомый узел не найден, а следовательно, не существует. Метод сообщает об этом факте, возвращая null.

Узел успешно найден

Если условие цикла while нарушено, то есть выполнение продолжается после тела цикла, поле iData объекта current равно key; это означает, что искомый узел был успешно найден. Метод возвращает узел, чтобы код, вызвавший find(), смог обратиться к полям этого узла.

Эффективность поиска по дереву

Как видно из описания, время поиска узла зависит от количества уровней. В приложении Workshop дерево может содержать до 31 узла, но не более пяти уровней, следовательно, для поиска любого узла требуется не более пяти сравнений. Это время $O(\log N)$, а более конкретно $O(\log_2 N)$ (логарифм по основанию 2). Тема будет более подробно рассмотрена ближе к концу главы.

Вставка узла

Чтобы вставить узел, необходимо сначала найти место для его вставки. Этот процесс почти эквивалентен поиску несуществующего узла (см. выше подраздел «Неудачный поиск»). Метод отслеживает узел от корня до узла, который станет родителем нового узла. Когда родитель будет найден, новый узел вставляется как левый или правый потомок в зависимости от того, будет ли ключ нового узла меньше или больше родительского ключа.

Вставка узла в приложении Workshop

Чтобы вставить новый узел в приложении Workshop, щелкните на кнопке `Ins`. Вам будет предложено ввести значение ключа вставляемого узла. Допустим, вы хотите вставить новый узел со значением 45; введите это число в текстовом поле.

Процедура вставки узла начинается с поиска позиции для вставки. На рис. 8.8, *а* показано, как это происходит.

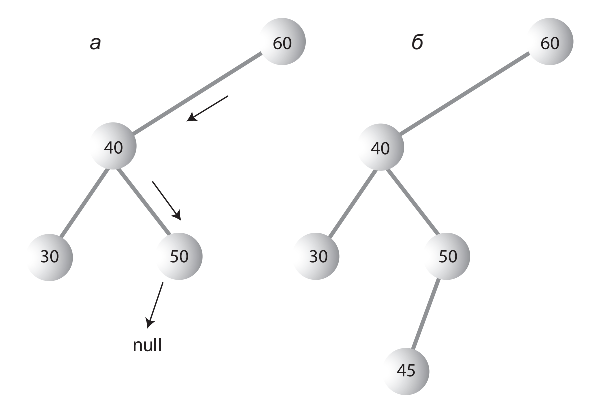


Рис. 8.8. Вставка узла: а — до вставки; б — после вставки

Значение 45 меньше 60, но больше 40 — переходим к узлу 50. Теперь нужно двигаться налево, потому что 45 меньше 50, но у узла 50 нет левого потомка; его поле `leftChild` равно `null`. При обнаружении `null` алгоритм вставки считает, что место для присоединения нового узла найдено. Приложение создает новый узел с ключом 45 (и случайно выбранным цветом) и связывает его с узлом 50 в качестве левого потомка, как показано на рис. 8.8, *б*.

Реализация вставки на языке Java

Сначала метод `insert()` создает новый узел на основе данных, переданных в аргументах. Далее он должен определить место для вставки нового узла. Используется примерно такой же код, как при поиске узла (см. раздел «Реализация поиска узла

на языке Java»). Различие в том, что при простом поиске обнаружение несуществующего узла (null) означает, что искомый узел не существует, и приводит к немедленному возвращению управления. При вставке перед возвращением управления узел включается в дерево (если потребуется, программа сначала создает его).

Искомым значением является элемент, передаваемый в аргументе id. Цикл while использует условие true, потому что обнаружение узла с тем же значением, что у id, игнорируется; узел с совпадающим ключом интерпретируется так, как если бы его ключ был больше искомого. (Мы вернемся к теме совпадения ключей позднее в этой главе.)

Место для вставки нового узла всегда находится успешно (если в системе хватит памяти); когда это произойдет, новый узел включается в дерево, а выполнение цикла while завершается командой return.

Код функции insert():

```
public void insert(int id, double dd)
   {
   Node newNode = new Node();    // Создание нового узла
   newNode.iData = id;           // Вставка данных
   newNode.dData = dd;
   if(root==null)                // Корневой узел не существует
      root = newNode;
   else                          // Корневой узел занят
      {
      Node current = root;       // Начать с корневого узла
      Node parent;
      while(true)                // (Внутренний выход из цикла)
         {
         parent = current;
         if(id < current.iData)  // Двигаться налево?
            {
            current = current.leftChild;
            if(current == null)  // Если достигнут конец цепочки
               {                 // вставить слева
               parent.leftChild = newNode;
               return;
               }
            }
         else                    // Или направо?
            {
            current = current.rightChild;
            if(current == null)  // Если достигнут конец цепочки,
               {                 // вставить справа
               parent.rightChild = newNode;
               return;
               }
            }
         }
      }
   }
// -------------------------------------------------------------
```

В новой переменной `parent` (родитель `current`) хранится последний отличный от `null` узел, посещенный при переборе (50 на рис. 8.8). Хранение этого узла необходимо, так как для проверки того, что предыдущее значение `current` не имело подходящего потомка, `current` присваивается `null`. Не сохранив `parent`, метод потеряет текущую позицию в дереве.

Вставка нового узла осуществляется изменением соответствующего указателя на потомка в `parent` (последний посещенный узел, отличный от `null`). В случае безуспешного поиска левого потомка `parent` новый узел присоединяется к `parent` как левый потомок, а при поиске правого потомка — соответственно как правый потомок. На рис. 8.8 значение 45 присоединяется к узлу 50 как левый потомок.

Обход дерева

Обходом дерева называется посещение всех его узлов в определенном порядке. На практике обход используется не так часто, как поиск, вставка и удаление узлов. Одна из причин заключается в том, что алгоритмы обхода не отличаются быстротой. Однако обход дерева бывает полезным в некоторых обстоятельствах, и он представляет интерес с теоретической точки зрения. (Вдобавок обход проще удаления, которым мы займемся в последнюю очередь.)

Существуют три простых алгоритма обхода дерева: *прямой* (preorder), *симметричный* (inorder) и *обратный* (postorder). Для деревьев двоичного поиска чаще всего применяется алгоритм симметричного обхода, поэтому сначала мы проанализируем его, а потом в общих чертах рассмотрим два других.

Симметричный обход

При симметричном обходе двоичного дерева все узлы перебираются в порядке возрастания ключей. Если вам потребуется создать отсортированный список данных двоичного дерева — это одно из возможных решений.

Простейший способ обхода основан на использовании рекурсии (см. главу 6, «Рекурсия»). При вызове рекурсивного метода для обхода всего дерева в аргументе передается узел. В исходном состоянии этим узлом является корень дерева. Метод должен выполнить только три операции:

1. Вызов самого себя для обхода левого поддерева узла.
2. Посещение узла.
3. Вызов самого себя для обхода правого поддерева узла.

Не забудьте, что *посещение* узла подразумевает выполнение некоторой операции: вывод данных, запись в файл и т. д.

Обход работает с любым двоичным деревом, не только с деревьями двоичного поиска. Алгоритм обхода не обращает внимания на значения ключей; его интересует только наличие у узла потомков.

Реализация обхода на языке Java

Код симметричного обхода дерева настолько прост, что мы приведем его еще до рассмотрения обхода в приложении Workshop. Метод `inOrder()` выполняет три операции, упоминавшиеся ранее. Посещение узла сводится к отображению его содержимого. Как и любая рекурсивная функция, метод должен обладать базовым ограничением — условием, при котором он немедленно возвращает управление без рекурсивного вызова. В методе `inOrder()` это происходит при передаче аргумента `null`. Код метода `inOrder()`:

```
private void inOrder(node localRoot)
   {
   if(localRoot != null)
      {
      inOrder(localRoot.leftChild);

      System.out.print(localRoot.iData + " ");
      inOrder(localRoot.rightChild);
      }
   }
```

В исходном вызове метода в аргументе передается корневой узел `root`:

```
inOrder(root);
```

Далее метод действует самостоятельно, рекурсивно вызывая самого себя до тех пор, пока не останется узлов для обхода.

Обход дерева из трех узлов

Простой пример дает представление о работе симметричного обхода. Предположим, вам потребовалось обойти дерево (рис. 8.9), которое состоит всего из трех узлов: корня (A) с левым (B) и правым (C) потомками.

Сначала метод `inOrder()` вызывается с аргументом A (корневой узел) — назовем эту версию вызова `inOrder(A)`. `inOrder(A)` сначала вызывает `inOrder()` для своего левого потомка `B` (второй вызов `inOrder()` будет обозначаться `inOrder(B)`).

Теперь `inOrder(B)` вызывает `inOrder()` для своего левого потомка. Однако левого потомка у узла B нет, поэтому соответствующий аргумент равен `null`. Созданный при этом вызов `inOrder()` назовем `inOrder(null)`. В итоге на текущий момент существуют три экземпляра `inOrder()`: `inOrder(A)`, `inOrder(B)` и `inOrder(null)`. Однако как только `inOrder(null)` обнаруживает, что аргумент равен `null`, он немедленно возвращает управление.

Теперь `inOrder(B)` переходит к посещению узла B; будем считать, что это означает вывод его данных. `inOrder(B)` снова вызывает `inOrder()` для своего правого потомка. Этот аргумент также равен `null`, поэтому второй вызов `inorder(null)` тоже немедленно возвращает управление. Соответственно вызов `inOrder(B)` выполнил шаги 1, 2 и 3, поэтому он возвращает управление (и перестает существовать).

Теперь управление возвращается вызову `inOrder(A)` в точку непосредственно после обхода левого потомка A. Метод посещает A, а затем снова вызывает `inOrder()`

с аргументом C, создавая вызов inOrder(C). Как и inOrder(B), inOrder(C) не имеет дочерних узлов, поэтому шаг 1 возвращает управление без выполнения каких-либо действий, на шаге 2 посещается узел C, а шаг 3 тоже возвращает управление. Выполнение inOrder(A) на этом прекращается, вызов возвращает управление, а обход завершается.

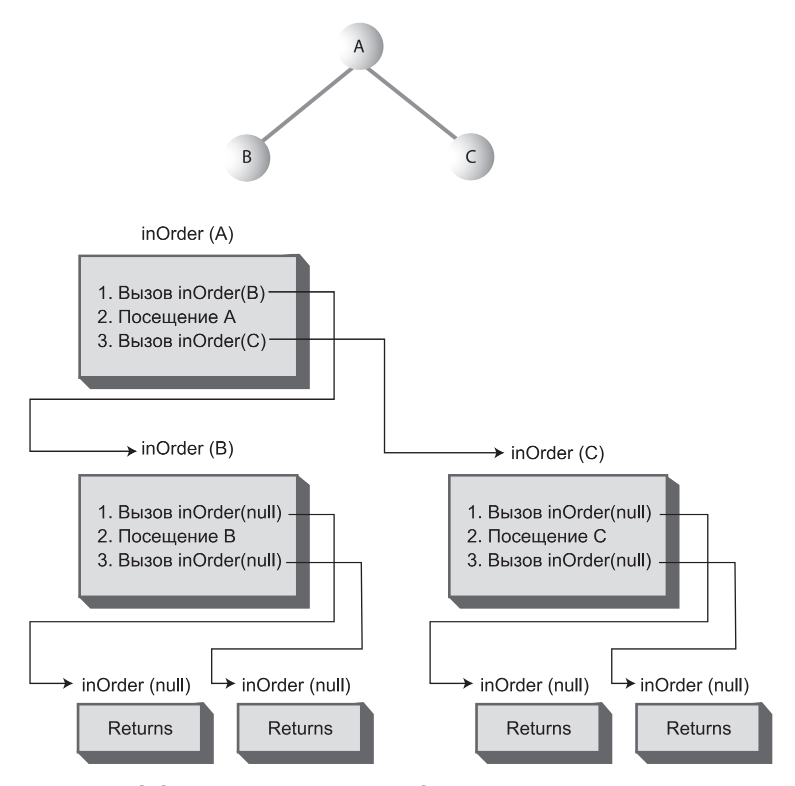


Рис. 8.9. Применение метода inOrder() к дереву из трех узлов

Узлы дерева в процессе обхода посещались последовательно, в порядке A, B, C. В дереве двоичного поиска этот порядок соответствует порядку возрастания ключей.

Более сложные деревья обрабатываются аналогичным образом. Метод inOrder() вызывает себя для каждого узла до тех пор, пока не переберет все узлы дерева.

Обход дерева в приложении Workshop

Чтобы проследить за выполнением обхода в приложении Workshop, многократно нажимайте кнопку Trav. (Никакие числа вводить не нужно.)

На рис. 8.10 показано, что происходит при обходе дерева в приложении Tree Workshop. Ситуация чуть сложнее, чем для дерева с тремя узлами из предыдущего примера. Красная стрелка начинает перемещение от корня дерева. В таблице 8.1

приведена последовательность ключей и соответствующих им сообщений. В приложении Workshop последовательность ключей отображается в нижней части области вывода.

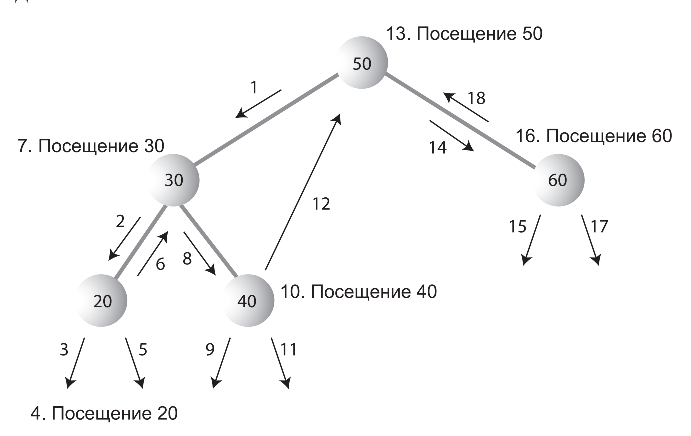


Рис. 8.10. Обход дерева в инфиксном порядке

Таблица 8.1. Обход узлов в приложении Tree Workshop

Номер шага	Положение красной стрелки (узел)	Сообщение	Список посещенных узлов
1	50 (корень)	Will check left child	
2	30	Will check left child	
3	20	Will check left child	
4	20	Will visit this node	
5	20	Will check right child	20
6	20	Will go to root of previous subtree	20
7	30	Will visit this node	20
8	30	Will check right child	20 30
9	40	Will check left child	20 30
10	40	Will visit this node	20 30
11	40	Will check right child	20 30 40
12	40	Will go to root of previous subtree	20 30 40
13	50	Will visit this node	20 30 40
14	50	Will check right child	20 30 40 50
15	60	Will check left child	20 30 40 50
16	60	Will visit this node	20 30 40 50
17	60	Will check right child	20 30 40 50 60
18	60	Will go to root of previous subtree	20 30 40 50 60
19	50	Done traversal	20 30 40 50 60

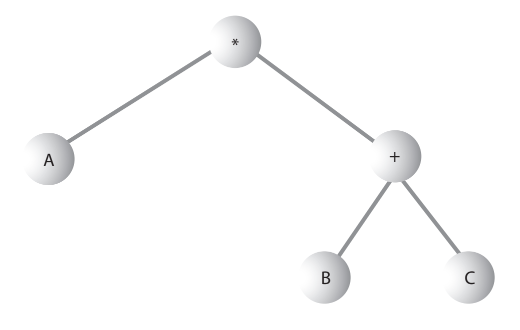
Для каждого узла алгоритм обходит левое поддерево, посещает сам узел и обходит правое поддерево. Например, для узла 30 эти действия выполняются на шагах 2, 7 и 8.

Алгоритм обхода не так сложен, как может показаться на первый взгляд. Чтобы понять, что происходит в процессе обхода, достаточно поэкспериментировать с несколькими разными деревьями в приложении Workshop.

Симметричный и обратный обход

Кроме симметричного обхода, также существует два других алгоритма обхода дерева, называемые прямым и обратным. Нетрудно представить, для каких целей может использоваться обход дерева в симметричном порядке, а польза от прямого и обратного перебора не столь очевидна. Тем не менее эти способы обхода находят применение при разборе или анализе алгебраических выражений. Остановимся на этой теме чуть подробнее.

Двоичное дерево (но не дерево двоичного поиска!) может использоваться для представления алгебраических выражений с бинарными операторами +, –, / и *. В корневом узле хранится оператор, а в других узлах — имя переменной (A, B или C) или другой оператор. Каждое поддерево представляет действительное алгебраическое выражение.



Инфиксная запись: A*(B+C)
Префиксная запись: *A+BC
Постфиксная запись: ABC+*

Рис. 8.11. Дерево, представляющее алгебраическое выражение

Например, двоичное дерево на рис. 8.11 представляет алгебраическое выражение

```
A*(B+C)
```

Это пример так называемой *инфиксной* записи, обычно используемой в алгебре. (За дополнительной информацией об инфиксной и постфиксной записи обращайтесь к разделу «Разбор арифметических выражений» главы 4, «Стеки и очереди».) При обходе дерева в симметричном порядке генерируется правильная инфиксная запись вида A * B + C, но круглые скобки вам придется вставить самостоятельно.

Какое отношение все это имеет к прямому и обратному обходу? В этих алгоритмах обхода используются те же три шага, но выполняются они в другой последовательности. Для метода прямого обхода `preorder()` последовательность выглядит так:

1. Посещение узла.
2. Вызов самого себя для обхода левого поддерева узла.
3. Вызов самого себя для обхода правого поддерева узла.

Для дерева на рис. 8.11 при обходе в прямом порядке будет сгенерировано выражение

```
*A+BC
```

Такая запись называется *префиксной*. Одно из ее преимуществ заключается в том, что в ней никогда не нужны круглые скобки; выражение полностью однозначно и без них. От начала выражения каждый оператор применяется к следующим двум операндам. Для первого оператора * это операнды A и +BC; для второго оператора + — B и C, так что последнее выражение в инфиксной записи имеет вид B + C. Вставляя его в исходное выражение * A + BC (префиксная запись), мы получаем A * (B + C) в инфиксной записи. Разные порядки обхода дерева помогают преобразовать алгебраическое выражение из одной формы в другую.

В третьем алгоритме обхода — *обратном* — метод выполняет те же шаги в иной последовательности:

1. Вызов самого себя для обхода левого поддерева узла.
2. Вызов самого себя для обхода правого поддерева узла.
3. Посещение узла.

Для дерева на рис. 8.11 при посещении узлов с обратным порядком обхода генерируется выражение

```
ABC+*
```

Такая запись называется *постфиксной*. Она означает следующее: «Применить завершающий оператор выражения * к первому и второму операндам». Первым операндом здесь является A, а вторым — BC+. Запись BC+ означает «Применить завершающий оператор выражения + к первому и второму операндам». Первым операндом здесь является B, вторым C; в инфиксной записи выполняемая операция принимает вид (B + C). Вставка в исходное выражение ABC + * (в постфиксной записи) дает A * (B + C) в инфиксной записи.

ПРИМЕЧАНИЕ

В листинг 8.1 включены методы как для симметричного, так и прямого и обратного обхода узлов.

Мы не будем подробно рассматривать эту тему, но при желании можно легко построить дерево, аналогичное изображенному на рис. 8.11, на основании постфиксного выражения. Это делается примерно так же, как при вычислении результата постфиксного выражения в программе `postfix.java` (см. листинг 4.8 в главе 4) —

только вместо операндов в стеке сохраняются целые поддеревья. Действия, выполняемые при обнаружении операнда:

1. Создать дерево с единственным узлом, содержащим операнд.
2. Занести дерево в стек.

Действия при обнаружении оператора:

1. Извлечь из стека два дерева операндов B и C.
2. Создать новое дерево A, корнем которого является оператор.
3. Присоединить B в качестве левого потомка A.
4. Присоединить C в качестве левого потомка A.
5. Занести полученное дерево обратно в стек.

Когда обработка постфиксной строки будет завершена, остается извлечь из стека один оставшийся элемент. Как ни удивительно, этот элемент будет содержать полное дерево, описывающее алгебраическое выражение. Чтобы получить префиксное и инфиксное представление исходной постфиксной записи, выполните обход дерева так, как описано выше. Реализация этого процесса предоставляется читателю в качестве упражнения.

Поиск минимума и максимума

Поиск минимального и максимального значения в дереве двоичного поиска выполняется почти тривиально. Операция выполняется настолько просто, что мы не включили ее ни в приложение Workshop, ни в листинг 8.1. И все же важно понимать, как это делается.

Чтобы получить минимальное значение ключа в дереве, перейдите от корня к левому потомку; затем перейдите к левому потомку этого потомка и т. д., пока не доберетесь до узла, не имеющего левого потомка. Этот узел и содержит минимальное значение ключа (рис. 8.12).

Следующий метод возвращает узел с минимальным значением ключа:

```
public Node minimum()     // Возвращает узел с минимальным ключом
   {
   Node current, last;
   current = root;              // Обход начинается с корневого узла
   while(current != null)       // и продолжается до низа
      {
      last = current;                 // Сохранение узла
      current = current.leftChild;    // Переход к левому потомку
      }
   return last;
   }
```

Минимальное значение пригодится при удалении узла.

Поиск максимума в дереве выполняется аналогичным образом, но вместо перехода налево следует переходить от правого потомка к правому потомку, пока не будет найден узел, не имеющий правого потомка. Этот узел и содержит максимальное

значение ключа. Код метода выглядит точно так же, не считая того, что последняя команда в цикле принимает вид

```
current = current.rightChild;  // Переход к правому потомку
```

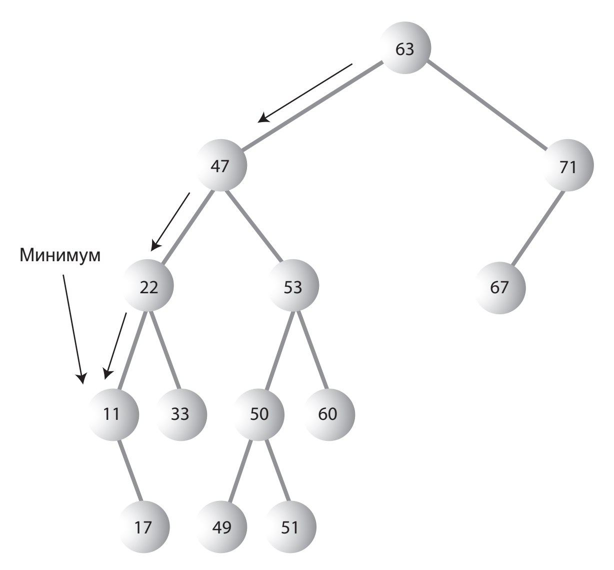


Рис. 8.12. Поиск минимума по дереву

Удаление узла

Удаление узлов является самой сложной из стандартных операций с деревьями двоичного поиска. Тем не менее удаление играет важную роль во многих приложениях, работающих с деревьями, а его подробное изучение помогает глубже разобраться в теме.

Удаление начинается с поиска удаляемого узла — процедура поиска уже была продемонстрирована ранее в методах `find()` и `insert()`. Когда узел будет найден, необходимо рассмотреть три возможных случая:

1. Удаляемый узел является листовым (не имеет потомков).
2. Удаляемый узел имеет одного потомка.
3. Удаляемый узел имеет двух потомков.

Мы последовательно разберем эти три случая. Первый совсем прост; второй почти так же прост; с третьим дело обстоит сложнее.

Случай 1. Удаляемый узел не имеет потомков

Чтобы удалить листовой узел, достаточно изменить поле соответствующего потомка в родительском узле, сохранив в нем `null` вместо ссылки на узел. Узел продолжает существовать, но перестает быть частью дерева (рис. 8.13).

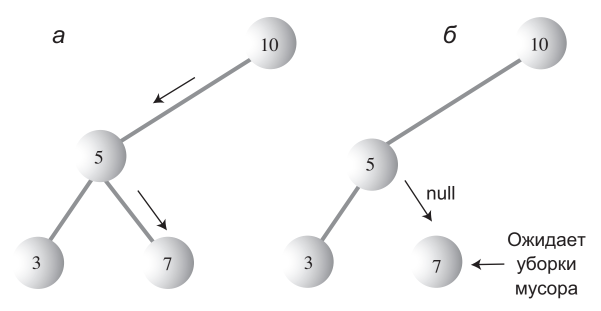


Рис. 8.13. Удаление узла, не имеющего потомков: *а* — до удаления; *б* — после удаления

Благодаря механизму уборки мусора, используемому в Java, вам не нужно беспокоиться об уничтожении самого узла. Когда уборщик мусора поймет, что программа не содержит ни одной ссылки на узел, последний будет удален из памяти. (В языках C и C++ вам пришлось бы выполнить команду `free()` или `delete()` для удаления узла из памяти.)

Удаление листового узла в приложении Workshop

Допустим, вы собираетесь удалить узел 7 на рис. 8.13. Щелкните на кнопке `Del` и введите число 7. Естественно, перед удалением узел должен быть найден; при повторных нажатиях `Del` маркер переходит от 10 и 5 к 7. Найденный узел удаляется без каких-либо последствий.

Удаление листового узла на языке Java

Начало метода `delete()` почти не отличается от методов `find()` и `insert()`. Удаляемый узел необходимо прежде всего найти. Как и в случае с `insert()`, необходимо запомнить родителя удаляемого узла для изменения полей его потомков. Если узел будет найден, цикл `while` прерывается; при этом `parent` содержит удаляемый узел. Если поиск оказался неудачным, `delete()` просто возвращает значение `false`.

```
public boolean delete(int key) // Удаление узла с заданным ключом
   {                           // (предполагается, что дерево не пусто)
   Node current = root;
   Node parent = root;
   boolean isLeftChild = true;

   while(current.iData != key)        // Поиск узла
      {
      parent = current;
      if(key < current.iData)         // Двигаться налево?
         {
         isLeftChild = true;
         current = current.leftChild;
         }
```

```
    else                            // Или направо?
       {
       isLeftChild = false;
       current = current.rightChild;
       }
    if(current == null)             // Конец цепочки
       return false;                // Узел не найден
    }
 // Удаляемый узел найден
 // Продолжение...
 }
```

Когда узел будет найден, мы сначала убеждаемся в том, что у него нет потомков. Если условие выполнено, проверяется особый случай корневого узла. Если удалямый узел является корневым, мы просто присваиваем ему `null`, что приводит к полной очистке дерева. В противном случае полю `leftChild` или `rightChild` родителя присваивается `null`, чтобы отсоединить узел от родителя.

```
// Продолжение delete()...
// Если узел не имеет потомков, он просто удаляется.
if(current.leftChild==null &&
                             current.rightChild==null)
   {
   if(current == root)             // Если узел является корневым,
      root = null;                 // дерево очищается
   else if(isLeftChild)
      parent.leftChild = null;     // Узел отсоединяется
   else                            // от родителя
      parent.rightChild = null;
   }
// Продолжение...
```

Случай 2. Удаляемый узел имеет одного потомка

Второй случай тоже обходится без особых сложностей. Узел имеет только две связи: с родителем и со своим единственным потомком. Требуется «вырезать» узел из этой цепочки, соединив родителя с потомком напрямую. Для этого необходимо изменить соответствующую ссылку в родителе (`leftChild` или `rightChild`), чтобы она указывала на потомка удаляемого узла. Ситуация показана на рис. 8.14.

Удаление узла с одним потомком в приложении Workshop

Допустим, приложение Workshop используется для работы с деревом на рис. 8.14. Требуется удалить узел 71, у которого есть только левый потомок, но нет правого. Щелкните на кнопке `Del` и введите 71. Продолжайте нажимать кнопку `Del`, пока стрелка не перейдет к 71. У узла 71 имеется только один потомок 63. Неважно, есть ли у узла 63 свои потомки или нет (в нашем примере у него есть один потомок 67).

Следующее нажатие Del приводит к удалению узла 71. Его место занимает левый потомок 63. Все поддерево, корнем которого был узел 63, перемещается вверх, а узел 63 становится новым правым потомком 52.

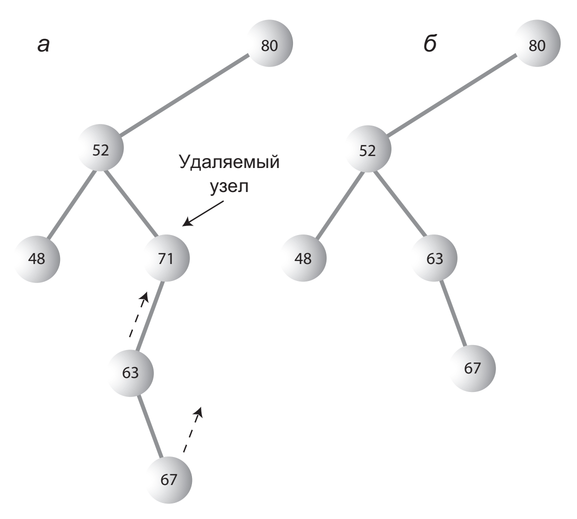


Рис. 8.14. Удаление узла с одним потомком: *а* — до удаления; *б* — после удаления

Попробуйте сгенерировать новое дерево с узлами, имеющими одного потомка, и проследите за тем, что происходит при удалении таких узлов. Обратите внимание на поддерево, корнем которого является потомок удаленного узла. Каким бы сложным ни было само дерево, оно просто перемещается вверх и связывается с родителем удаленного узла, становясь его новым потомком.

Удаление узла с одним потомком на языке Java

Следующий код демонстрирует обработку ситуации с одним потомком. Возможны четыре варианта: потомок удаляемого узла может быть левым или правым, и в каждом из этих случаев удаляемый узел может быть левым или правым потомком своего родителя.

Также возможен особый случай: если удаляется корневой узел, не имеющий родителя, он просто заменяется соответствующим поддеревом. Приведенный ниже код следует за кодом удаления листовых узлов из предыдущего подраздела:

```
// Продолжение delete()...
// Если нет правого потомка, узел заменяется левым поддеревом
else if(current.rightChild==null)
   if(current == root)
      root = current.leftChild;
   else if(isLeftChild)           // Левый потомок родителя
      parent.leftChild = current.leftChild;
   else                           // Правый потомок родителя
      parent.rightChild = current.leftChild;
```

```
// Если нет левого потомка, узел заменяется правым поддеревом
else if(current.leftChild==null)
   if(current == root)
      root = current.rightChild;
   else if(isLeftChild)           // Левый потомок родителя
      parent.leftChild = current.rightChild;
   else                           // Правый потомок родителя
      parent.rightChild = current.rightChild;
// Продолжение...
```

Обратите внимание, насколько ссылки упрощают удаление целых поддеревьев. Для этого достаточно отсоединить старую ссылку от поддерева и создать новую ссылку на него в другом месте. Хотя поддерево может содержать большое количество узлов, вам не приходится удалять их по отдельности. Более того, узлы «перемещаются» только в том смысле, что концептуально они занимают новое положение по отношению к другим узлам. С точки зрения программы изменилась только ссылка на корень поддерева.

Случай 3. Удаляемый узел имеет двух потомков

А теперь начинается самое интересное. Если удаляемый узел имеет двух потомков, нельзя просто заменить его одним из этих потомков (по крайней мере если потомок имеет собственных потомков). Почему? Взгляните на рис. 8.15 и представьте, как удаляемый узел 25 заменяется своим правым поддеревом с корнем 35. Какой узел должен быть левым потомком узла 35? Левый потомок удаленного узла 15 или левым потомком нового узла 30? В обоих случаях узел 30 окажется не на своем месте, но и просто выбросить его из дерева тоже нельзя.

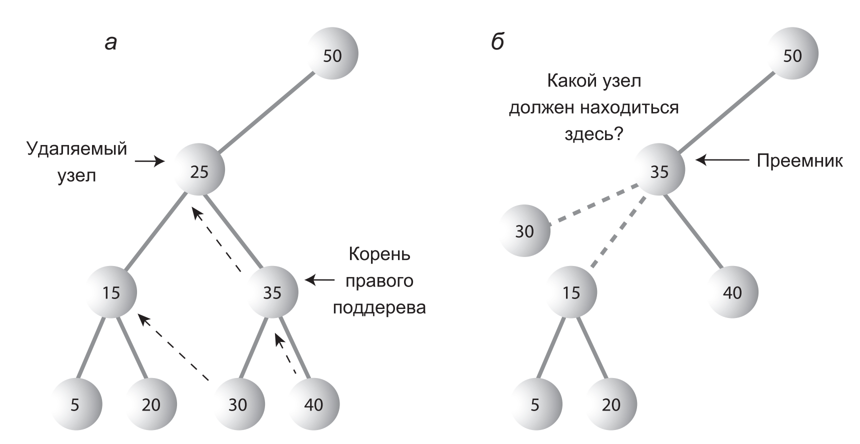


Рис. 8.15. Простая замена поддеревом невозможна: *а* — до удаления; *б* — после удаления

К счастью, существует полезный прием, который поможет нам справиться с проблемой. С другой стороны, даже при использовании этого приема приходится учитывать множество особых случаев. Вспомните, что мы работаем с деревом дво-

ичного поиска, в котором узлы располагаются в порядке возрастания ключей. Для каждого узла узел со следующим по величине ключом называется его *преемником*. Так, на рис. 8.15, *а* узел 30 является преемником узла 25.

Итак, обещанный прием: чтобы удалить узел с двумя потомками, *замените его преемником*. На рис. 8.16 изображена замена удаленного узла преемником. Обратите внимание на то, что порядок узлов при этом не нарушился. (Это еще не все, если у самого преемника имеются потомки — вскоре мы вернемся к этой возможности.)

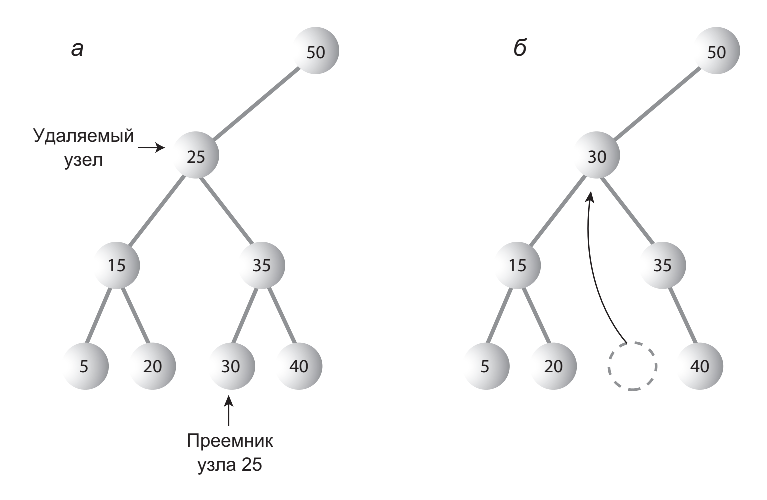


Рис. 8.16. Замена узла преемником: *а* — до удаления; *б* — после удаления

Поиск преемника

Как найти преемника узла? Человек сделает это очень быстро (по крайней мере для небольших деревьев) — ему достаточно взглянуть на дерево и найти число, следующее по величине за ключом удаляемого узла. На рис. 8.16 сразу видно, что преемником узла 25 является узел 30. В дереве нет другого числа, большего 25 и меньшего 35. Однако компьютер «взглянуть» не может; ему нужен алгоритм.

Сначала программа переходит к правому потомку исходного узла, ключ которого должен быть больше ключа узла. Затем она переходит к левому потомку правого потомка (если он существует), к левому потомку левого потомка и т. д., следуя вниз по цепочке левых потомков. Последний левый потомок на этом пути является преемником исходного узла (рис. 8.17).

Как работает этот алгоритм? Фактически мы ищем *наименьший узел* в наборе узлов, *больших исходного узла*. В поддереве правого потомка исходного узла все узлы больше исходного узла, что следует из самого определения дерева двоичного поиска. В этом дереве ищется наименьшее значение. Как говорилось выше, минимальный узел поддерева находится отслеживанием пути, состоящего из левых потомков. Таким образом, алгоритм находит минимальное значение, большее исходного узла, которое и является преемником удаляемого узла в соответствии с определением.

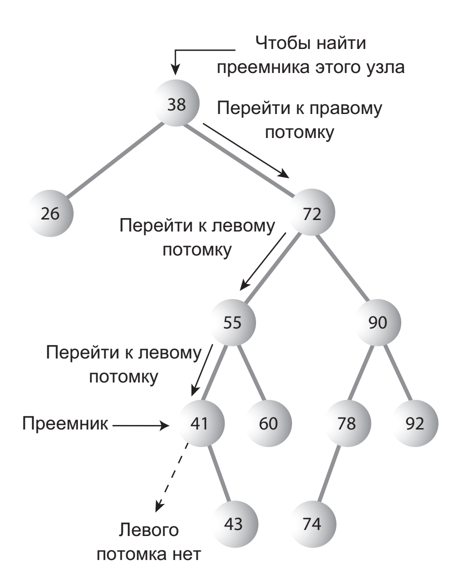


Рис. 8.17. Поиск преемника

Если у правого потомка исходного узла нет левых потомков, то сам правый потомок становится преемником (рис. 8.18).

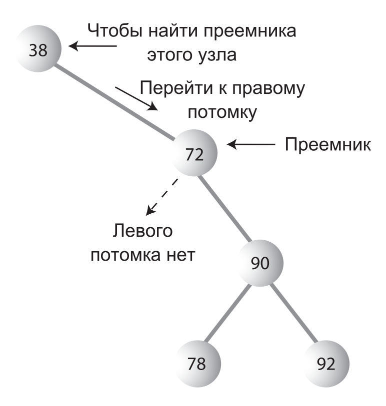


Рис. 8.18. Правый потомок становится преемником

Удаление узла с двумя потомками в приложении Workshop

Сгенерируйте дерево в приложении Workshop и выберите узел с двумя потомками. Теперь выясните, какой узел является его преемником — перейдите к правому потомку, а затем проследите цепочку левых потомков (если они есть). Также стоит убедиться в том, что преемник не имеет своих потомков (наличие потомков усложняет ситуацию, так как перемещать приходится целые поддеревья вместо одного узла).

Выбрав узел для удаления, щелкните на кнопке Del. Вам будет предложено ввести ключ удаляемого узла. Когда ключ будет введен, при повторных нажатиях кнопки Del красная стрелка будет перемещаться по дереву к указанному узлу. Удаленный узел заменяется преемником.

Допустим, приложение Workshop используется для удаления узла с ключом 25 в примере, представленном ранее на рис. 8.15. Красная стрелка переходит от корня 50 к узлу 25, который затем заменяется значением 30.

Поиск преемника на языке Java

Ниже приведен фрагмент кода метода getSuccessor(), который возвращает преемника узла, переданного в аргументе delNode. (Метод подразумевает, что у delNode существует правый потомок, но это условие заведомо выполняется — предыдущая проверка определила, что удаляемый узел имеет двух потомков.)

```
// Метод возвращает узел со следующим значением после delNode.
// Для этого он сначала переходит к правому потомку, а затем
// отслеживает цепочку левых потомков этого узла.

private node getSuccessor(node delNode)
   {
   Node successorParent = delNode;
   Node successor = delNode;
   Node current = delNode.rightChild;   // Переход к правому потомку
   while(current != null)               // Пока остаются левые потомки
      {
      successorParent = successor;
      successor = current;
      current = current.leftChild;      // Переход к левому потомку
      }
                                        // Если преемник не является
   if(successor != delNode.rightChild)  // правым потомком,
      {                                 // создать связи между узлами
      successorParent.leftChild = successor.rightChild;
      successor.rightChild = delNode.rightChild;
      }
   return successor;
   }
```

Метод сначала переходит к правому потомку delNode, а затем в цикле while проходит по цепочке левых потомков этого правого потомка. При выходе из цикла while переменная successor содержит преемника delNode.

При обнаружении преемника также может возникнуть необходимость в обращении к его родителю, поэтому в цикле while также отслеживается родитель текущего узла.

Метод getSuccessor() также выполняет две дополнительные операции, помимо поиска преемника. Однако чтобы понять их смысл, необходимо немного отойти и взглянуть на «общую картину».

Как мы уже видели, узел-преемник может занимать одну из двух возможных позиций относительно удаляемого узла `current`: он может быть его правым потомком или входить в цепочку левых потомков его правого потомка. Рассмотрим каждую из этих ситуаций.

Преемник является правым потомком delNode

Если `successor` является правым потомком `current`, ситуация немного упрощается, потому что мы можем просто переместить все поддерево, корнем которого является преемник, и вставить его на место удаленного узла. Эта операция выполняется всего за два шага:

1. Отсоединить `current` от поля `rightChild` (или `leftChild`) его родителя. Сохранить в поле ссылку на преемника.
2. Отсоединить левого потомка `current` от `current` и сохранить ссылку на него в поле `leftChild` объекта `successor`.

В коде `delete()` эти операции выполняются следующими командами:

```
1. parent.rightChild = successor;
2. successor.leftChild = current.leftChild;
```

На рис. 8.19 показано, как изменяются ссылки при выполнении этих двух команд.

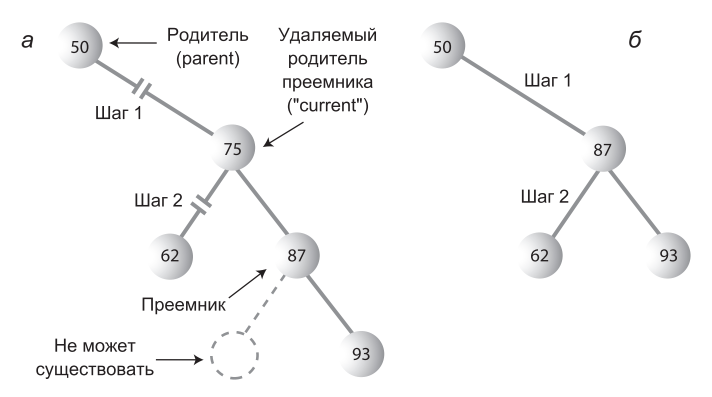


Рис. 8.19. Удаление в случае, когда преемник является правым потомком: *а* — до удаления; *б* — поле удаления

Вот как выглядит код в контексте (продолжение цепочки `else-if`, приведенной ранее):

```
// Продолжение delete()
else  // Два потомка, узел заменяется преемником
   {
   // Поиск преемника для удаляемого узла (current)
   Node successor = getSuccessor(current);

   // Родитель current связывается с посредником
   if(current == root)
```

```
      root = successor;
   else if(isLeftChild)
      parent.leftChild = successor;
   else
      parent.rightChild = successor;
   // Преемник связывается с левым потомком current
   successor.leftChild = current.leftChild;
   }  // Конец else для двух потомков
// (преемник не может иметь левого потомка)
return true;
}  // Конец метода delete()
```

Обратите внимание: мы (наконец-то) добрались до конца метода `delete()`. Проанализируем код этих двух операций:

- Шаг 1: если удаляемый узел `current` является корневым, то он не имеет родителя, поэтому `root` просто присваивается `successor`. В противном случае удаляемый узел может быть либо левым, либо правым потомком (на рис. 8.19 это правый потомок), поэтому в соответствующем поле его родителя сохраняется ссылка на `successor`. Когда `delete()` возвращает управление, а `current` выходит из области видимости, на узел, на который ссылается `current`, не остается ни одной ссылки, поэтому он будет уничтожен в ходе уборки мусора Java.
- Шаг 2: в поле левого потомка преемника сохраняется ссылка на левого потомка `current`.

Что произойдет, если у преемника имеются свои потомки? Прежде всего, узел преемника заведомо не имеет левого потомка. Это утверждение истинно в любом случае — и если преемник является правым потомком удаляемого узла, и если он является одним из левых потомков правого потомка. Откуда это известно?

Вспомните, что алгоритм поиска преемника сначала переходит к правому потомку, а затем перебирает всех его левых потомков. Перебор прекращается при достижении узла, не имеющего левого потомка, так что преемник не может иметь левых потомков в соответствии с алгоритмом. Если бы у него был левый потомок, то последний и стал бы преемником.

Проверьте истинность этого утверждения при помощи приложения Workshop. Сколько бы деревьев вы ни создавали, вам никогда не удастся найти ситуацию, при которой у преемника узла имеется левый потомок (предполагается, что исходный узел имеет двух потомков — собственно, эта ситуация и создает больше всего хлопот).

С другой стороны, ничто не мешает преемнику иметь правого потомка. Это не создает особых проблем, когда преемник является правым потомком удаляемого узла. При перемещении преемника его правое поддерево просто перемещается вместе с ним. С правым потомком удаляемого узла конфликтов тоже нет, потому что этим правым потомком является сам преемник.

Как будет показано в следующем разделе, если преемник не является правым потомком удаляемого узла, то правому потомку преемника придется уделить больше внимания.

Преемник входит в число левых потомков правого потомка delNode

Если преемник входит в число левых потомков правого потомка удаляемого узла, то удаление выполняется за четыре шага:

1. Сохранить ссылку на правого потомка преемника в поле `leftChild` родителя преемника.
2. Сохранить ссылку на правого потомка удаляемого узла в поле `rightChild` преемника.
3. Убрать `current` из поля `rightChild` его родителя и сохранить в этом поле ссылку на преемника `successor`.
4. Убрать ссылку на левого потомка `current` из объекта `current` и сохранить ее в поле `leftChild` объекта `successor`.

Шаги 1 и 2 выполняются методом `getSuccessor()`, а шаги 3 и 4 выполняются в `delete()`. На рис. 8.20 показано изменение ссылок в процессе их выполнения.

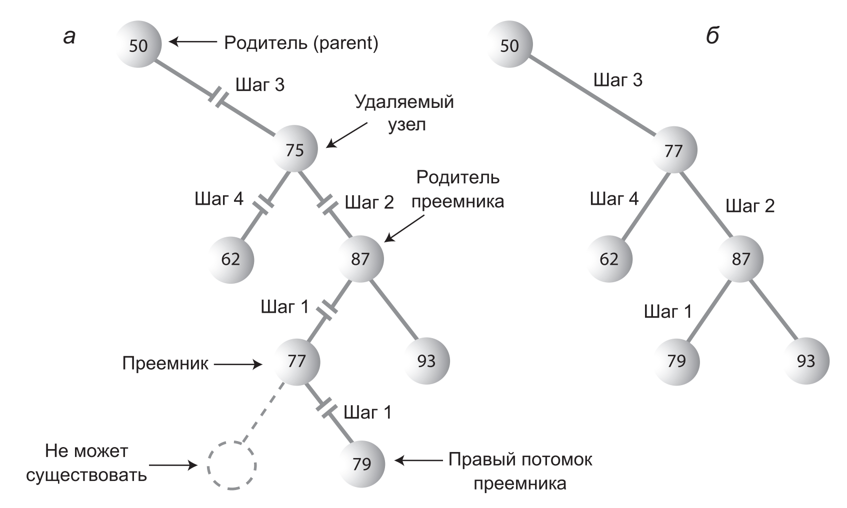


Рис. 8.20. Удаление в случае, когда преемник является левым потомком: *а* — до удаления; *б* — после удаления

А вот как выглядит код реализации этих четырех шагов:

```
1. successorParent.leftChild = successor.rightChild;
2. successor.rightChild = delNode.rightChild;
3. parent.rightChild = successor;
4. successor.leftChild = current.leftChild;
```

(На шаге 3 также может использоваться левый потомок `parent`.) Числа на рис. 8.20 обозначают связи, изменяемые на каждом из четырех шагов. Шаг 1 фактически заменяет преемника его правым поддеревом. Шаг 2 оставляет правого потомка удаляемого узла на положенном месте (это происходит автоматически,

когда преемник является правым потомком удаляемого узла). Шаги 1 и 2 выполняются в команде if, завершающей метод getSuccessor(). Еще раз приведем соответствующий фрагмент:

```
                                 // Если преемник не является
if(successor != delNode.rightChild)  // правым потомком,
   {                             // создать связи между узлами
   successorParent.leftChild = successor.rightChild;
   successor.rightChild = delNode.rightChild;
   }
return successor;
}
```

Эти действия удобнее выполнять здесь, чем в delete(), потому что в getSuccessor() мы можем легко определить местонахождение родителя преемника в ходе перемещения по дереву в процессе поиска преемника.

Шаги 3 и 4 мы уже видели; они совпадают с шагами 1 и 2 в случае, когда преемник является правым потомком удаляемого узла, а соответствующий код находится в команде if в конце delete().

Так ли уж необходимо удаление?

Если вы дочитали до этого момента, то уже знаете, что удаление узлов — весьма непростая операция. Она настолько сложна, что некоторые программисты предпочитают обходиться без нее. Они включают в класс node новое поле логического типа с именем вида isDeleted. Чтобы удалить узел, они просто присваивают этому полю значение true. Другие операции — такие, как find(), — прежде чем работать с узлом, проверяют это поле и убеждаются в том, что узел не помечен как удаленный. При таком подходе удаление узла не изменяет структуру дерева. Конечно, это также означает, что память может заполняться «удаленными» узлами.

Такой подход выглядит компромиссно, но он может оказаться подходящим при относительно небольшом количестве удалений из дерева. (Например, если данные бывших работников остаются в базе данных отдела кадров навсегда.)

Эффективность двоичных деревьев

Как мы уже видели, при выполнении многих операций с деревьями приходится спускаться от уровня к уровню для поиска определенного узла. Сколько времени занимает такой переход? В полном дереве на нижнем уровне находится около половины узлов. (А точнее, если дерево заполнено, на нижнем уровне находится на один узел больше, чем в остальном дереве.) Таким образом, около половины всех операций поиска, вставки или удаления требует поиска узла на нижнем уровне. (Еще четверть операций требует поиска на предпоследнем уровне и т. д.)

Алгоритм поиска посещает один узел на каждом уровне. Следовательно, чтобы оценить длительность выполнения операции, необходимо знать количество

уровней в дереве. Таблица 8.2 показывает, сколько уровней необходимо для хранения заданного количества узлов полного дерева.

Таблица 8.2. Количество уровней для заданного количества узлов

Количество узлов	Количество уровней
1	1
3	2
7	3
15	4
31	5
...	
1023	10
...	
32 767	15
...	
1 048 575	20
...	
33 554 432	25
...	
1 073 741 824	30

Ситуация напоминает упорядоченные массивы, рассматривавшиеся в главе 2. Тогда количество сравнений при двоичном поиске было примерно равно логарифму количества ячеек массива по основанию 2. В данном случае для количества узлов в первом столбце N и количества уровней во втором столбце L можно сказать, что N на 1 меньше 2 в степени L, или

$$N = 2^L - 1.$$

Прибавляя 1 к обоим сторонам уравнения, получаем

$$N + 1 = 2^L.$$

Или в эквивалентной записи

$$L = \log_2(N + 1).$$

Таким образом, время выполнения стандартных операций с деревом пропорционально логарифму N по основанию 2. В O-синтаксисе время выполнения таких операций обозначается $O(\log_2 N)$.

Для неполных деревьев анализ затрудняется. Можно лишь утверждать, что в неполном дереве с заданным количеством уровней среднее время поиска будет меньше, чем в полном дереве, потому что до нижних уровней доберется меньшее количество операций поиска.

Сравните дерево с другими структурами данных, рассматривавшимися до настоящего момента. В неупорядоченном массиве или связанном списке, содержащем 1 000 000 элементов, поиск нужного элемента требует в среднем 500 000 сравнений. Но для дерева с 1 000 000 узлов хватает 20 (и менее) сравнений.

В упорядоченном массиве элементы находятся быстро, но вставка потребует в среднем 500 000 перемещений. Вставка элемента в дерево с 1 000 000 узлов требует 20 и менее сравнений, а также незначительного времени на связывание узла.

Аналогичным образом, удаление элемента из массива с 1 000 000 элементов требует перемещения в среднем 500 000 элементов, тогда как удаление элемента из дерева с 1 000 000 узлов требует 20 и менее сравнений для поиска, также (возможно) еще нескольких сравнений для поиска преемника и незначительного времени для отсоединения элемента и присоединения преемника.

Таким образом, дерево обеспечивает высокую эффективность выполнения всех основных операций хранения данных. Обход выполняется не так быстро, как другие операции. Впрочем, для типичной большой базы данных обход вряд ли можно назвать типичной операцией. Обход дерева чаще используется при разборе алгебраических и других выражений, которые редко имеют большую длину.

Представление дерева в виде массива

Наши примеры кода основаны на представлении ребер дерева ссылками `leftChild` и `rightChild`, хранимыми в каждом узле. Однако наряду с таким представлением также существует совершенно другое представление дерева в виде массива.

В таком представлении узлы хранятся в массиве и не связываются ссылками. Позиция узла в массиве соответствует его положению в дереве. Элемент с индексом 0 представляет корень дерева, элемент с индексом 1 — его левого потомка и так далее, с перебором слева направо на каждом уровне дерева (рис. 8.21).

Каждая позиция в дереве (независимо от того, представляет она существующий узел или нет) соответствует определенной ячейке массива. Включение узла в некоторую позицию дерева означает вставку узла в соответствующую ячейку массива. Ячейки, представляющие пустые позиции дерева, заполняются нулями или `null`.

В такой схеме родитель и потомки узла вычисляются по простым формулам на основании индекса узла в массиве. Если индекс узла в массиве равен `index`, то индекс его левого потомка равен

```
2*index + 1
```

Индекс правого потомка:

```
2*index + 2
```

Индекс родителя:

```
(index-1) / 2
```

(оператор / обозначает целочисленное деление без остатка). Рисунок 8.21 поможет вам убедиться в истинности этих формул.

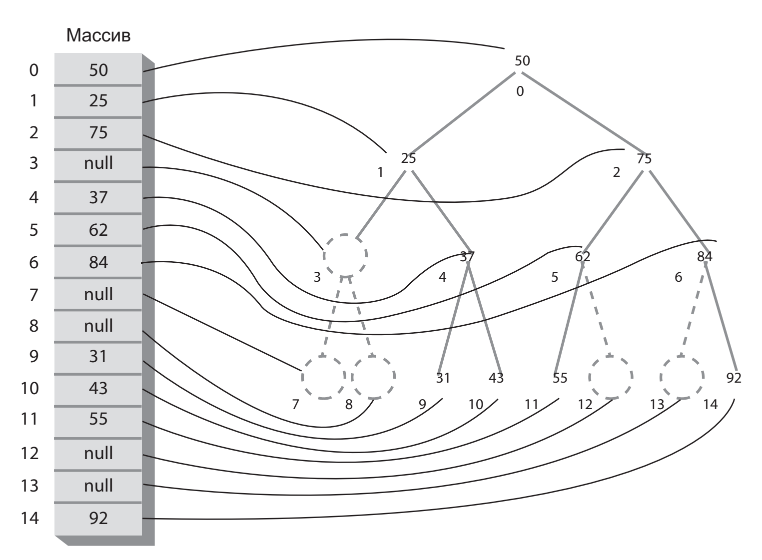


Рис. 8.21. Представление дерева в виде массива

В большинстве случаев эффективность представления дерева в виде массива оставляет желать лучшего. Незаполненные и удаленные узлы оставляют пустоты в массиве и, как следствие, приводят к неэффективному расходованию памяти. Еще хуже другое: если удаление узла требует перемещения поддеревьев, каждый узел поддерева приходится перемещать в новую ячейку массива; для больших деревьев эта операция занимает много времени. Но если удаление запрещено, представление дерева в виде массива может быть весьма полезным, особенно если динамическое выделение памяти для каждого узла по каким-то причинам занимает слишком много времени. Представление в виде массива также может пригодиться в особых ситуациях. Например, дерево из приложения Tree Workshop использует внутреннее представление в виде массива, упрощающее установление соответствия между узлами массива и фиксированными позициями области вывода.

Дубликаты ключей

При работе с деревьями (как и другими структурами данных) необходимо решить проблему дубликатов ключей. В приводившемся ранее коде `insert()` и в приложении Workshop узел с ключом-дубликатом вставляется в качестве правого потомка своего «близнеца».

Проблема в том, что метод `find()` находит только первый из двух (а возможно, и более) дубликатов. В коде `find()` можно реализовать проверку дополнительных элементов, чтобы метод различал элементы данных даже при совпадении ключей, но это (хотя бы в какой-то степени) снизит его эффективность.

Одно из возможных решений — простой запрет на дублирование ключей. Если дубликаты исключены самой природой данных (например, табельные номера ра-

ботников), то проблем не будет. В противном случае придется вносить изменения в метод `insert()`: проверять равенство ключей в процессе вставки и в случае обнаружения дубликата отменять вставку.

Кнопка `Fill` в приложении Workshop исключает дубликаты при генерировании случайных ключей.

Полный код программы tree.java

В этом разделе приводится полный код программы со всеми методами и фрагментами, рассматривавшимися ранее в этой главе. Также в программе реализован примитивный пользовательский интерфейс, который позволяет выбрать операцию (поиск, вставка, удаление, обход и вывод содержимого дерева) посредством ввода символов. Метод вывода строит изображение дерева в текстовом режиме вывода. На рис. 8.22 показан пример выходных данных программы.



Рис. 8.22. Вывод программы tree.java

На рисунке пользователь сначала ввел команду *s* (вывод содержимого дерева) и затем команду *i* и *48* (вставка узла с указанным ключом). Наконец, повторная команда *s* выводит дерево с новым узлом. Узел *48* отображается в нижней части области вывода.

Приложение поддерживает команды *s*, *i*, *f*, *d* и *t* соответственно для вывода, вставки, поиска, удаления и обхода. Команды *i*, *f* и *d* запрашивают ключ элемента, с которым выполняется операция. Команда *t* позволяет выбрать порядок обхода: *1* — прямой, *2* — симметричный и *3* — обратный. Значения ключей отображаются в выбранном порядке.

На экране значения ключей выстроены в форме, отдаленно напоминающей дерево; впрочем, ребра вам придется «дорисовать» силой воображения. Два дефиса (`--`)

представляют отсутствующий узел. Программа сначала создает несколько узлов, чтобы дерево содержало данные перед вставкой. Измените код инициализации, чтобы дерево в исходном состоянии содержало фиксированный набор узлов или было пустым.

Поэкспериментируйте с программой в листинге 8.1 так же, как вы экспериментировали с приложением Workshop. Однако в отличие от приложения Workshop, программа не выводит подробное описание действий при выполнении операции; все действия выполняются сразу.

Листинг 8.1. Программа tree.java

```
// tree.java
// Работа с двоичным деревом
// Запуск программы: C>java TreeApp
import java.io.*;
import java.util.*;              // Для использования класса Stack
////////////////////////////////////////////////////////////////
class Node
   {
   public int iData;              // Данные, используемые в качестве ключа
   public double dData;           // Другие данные
   public Node leftChild;         // Левый потомок узла
   public Node rightChild;        // Правый потомок узла

   public void displayNode()      // Вывод узла
      {
      System.out.print('{');
      System.out.print(iData);
      System.out.print(", ");
      System.out.print(dData);
      System.out.print("} ");
      }
   }  // Конец класса Node
////////////////////////////////////////////////////////////////
class Tree
   {
   private Node root;             // first node of tree

// -------------------------------------------------------------
   public Tree()                  // Конструктор
      { root = null; }            // Пока нет ни одного узла
// -------------------------------------------------------------
   public Node find(int key)      // Поиск узла с заданным ключом
      {                           // (предполагается, что дерево не пустое)
      Node current = root;               // Начать с корневого узла
      while(current.iData != key)        // Пока не найдено совпадение
         {
         if(key < current.iData)         // Двигаться налево?
            current = current.leftChild;
         else                            // Или направо?
```

```
            current = current.rightChild;
         if(current == null)             // Если потомка нет,
            return null;                 // поиск завершился неудачей
         }
      return current;                    // Элемент найден
      }
// -------------------------------------------------------------
   public void insert(int id, double dd)
      {
      Node newNode = new Node();    // Создание нового узла
      newNode.iData = id;           // Вставка данных
      newNode.dData = dd;
      if(root==null)                // Корневой узел не существует
         root = newNode;
      else                          // Корневой узел занят
         {
         Node current = root;       // Начать с корневого узла
         Node parent;
         while(true)                // (внутренний выход из цикла)
            {
            parent = current;
            if(id < current.iData)  // Двигаться налево?
               {
               current = current.leftChild;
               if(current == null)  // Если достигнут конец цепочки,
                  {                 // вставить слева
                  parent.leftChild = newNode;
                  return;
                  }
               }
            else                    // Или направо?
               {
               current = current.rightChild;
               if(current == null)  // Если достигнут конец цепочки,
                  {                 // вставить справа
                  parent.rightChild = newNode;
                  return;
                  }
               }
            }
         }
      }
// -------------------------------------------------------------
   public boolean delete(int key) // Удаление узла с заданным ключом
      {                           // (предполагается, что дерево не пусто)
      Node current = root;
      Node parent = root;
      boolean isLeftChild = true;

      while(current.iData != key)        // Поиск узла
```

продолжение ⇨

Листинг 8.1 (*продолжение*)

```
      {
      parent = current;
      if(key < current.iData)         // Двигаться налево?
         {
         isLeftChild = true;
         current = current.leftChild;
         }
      else                            // Или направо?
         {
         isLeftChild = false;
         current = current.rightChild;
         }
      if(current == null)             // Конец цепочки
         return false;                // Узел не найден
      }
   // Удаляемый узел найден

   // Если узел не имеет потомков, он просто удаляется.
   if(current.leftChild==null &&
                             current.rightChild==null)
      {
      if(current == root)             // Если узел является корневым,
         root = null;                 // дерево очищается
      else if(isLeftChild)
         parent.leftChild = null;     // Узел отсоединяется
      else                            // от родителя
         parent.rightChild = null;
      }

   // Если нет правого потомка, узел заменяется левым поддеревом
   else if(current.rightChild==null)
      if(current == root)
         root = current.leftChild;
      else if(isLeftChild)
         parent.leftChild = current.leftChild;
      else
         parent.rightChild = current.leftChild;

   // Если нет левого потомка, узел заменяется правым поддеревом
   else if(current.leftChild==null)
      if(current == root)
         root = current.rightChild;
      else if(isLeftChild)
         parent.leftChild = current.rightChild;
      else
         parent.rightChild = current.rightChild;

   else  // Два потомка, узел заменяется преемником
```

```
        {
        // Поиск преемника для удаляемого узла (current)
        Node successor = getSuccessor(current);

        // Родитель current связывается с посредником
        if(current == root)
           root = successor;
        else if(isLeftChild)
           parent.leftChild = successor;
        else
           parent.rightChild = successor;

        // Преемник связывается с левым потомком current
        return true;                     // Признак успешного завершения
     }
// -------------------------------------------------------------
   // Метод возвращает узел со следующим значением после delNode.
   // Для этого он сначала переходит к правому потомку, а затем
   // отслеживает цепочку левых потомков этого узла.
   private Node getSuccessor(Node delNode)
      {
      Node successorParent = delNode;
      Node successor = delNode;
      Node current = delNode.rightChild;   // Переход к правому потомку
      while(current != null)               // Пока остаются левые потомки
         {
         successorParent = successor;
         successor = current;
         current = current.leftChild;      // Переход к левому потомку
         }
                                           // Если преемник не является
      if(successor != delNode.rightChild)  // правым потомком,
         {                                 // создать связи между узлами
         successorParent.leftChild = successor.rightChild;
         successor.rightChild = delNode.rightChild;
         }
      return successor;
      }
// -------------------------------------------------------------
   public void traverse(int traverseType)
      {
      switch(traverseType)
         {
         case 1: System.out.print("\nPreorder traversal: ");
                 preOrder(root);
                 break;
         case 2: System.out.print("\nInorder traversal:  ");
                 inOrder(root);
                 break;
```

продолжение ➪

Листинг 8.1 (*продолжение*)

```
         case 3: System.out.print("\nPostorder traversal: ");
                 postOrder(root);
                 break;
         }
      System.out.println();
      }
// -------------------------------------------------------------
   private void preOrder(Node localRoot)
      {
      if(localRoot != null)
         {
         System.out.print(localRoot.iData + " ");
         preOrder(localRoot.leftChild);
         preOrder(localRoot.rightChild);
         }
      }
// -------------------------------------------------------------
   private void inOrder(Node localRoot)
      {
      if(localRoot != null)
         {
         inOrder(localRoot.leftChild);
         System.out.print(localRoot.iData + " ");
         inOrder(localRoot.rightChild);
         }
      }
// -------------------------------------------------------------
   private void postOrder(Node localRoot)
      {
      if(localRoot != null)
         {
         postOrder(localRoot.leftChild);
         postOrder(localRoot.rightChild);
         System.out.print(localRoot.iData + " ");
         }
      }
// -------------------------------------------------------------
   public void displayTree()
      {
      Stack globalStack = new Stack();
      globalStack.push(root);
      int nBlanks = 32;
      boolean isRowEmpty = false;
      System.out.println(
      "......................................................");
      while(isRowEmpty==false)
         {
         Stack localStack = new Stack();
         isRowEmpty = true;
```

```
         for(int j=0; j<nBlanks; j++)
            System.out.print(' ');

         while(globalStack.isEmpty()==false)
            {
            Node temp = (Node)globalStack.pop();
            if(temp != null)
               {
               System.out.print(temp.iData);
               localStack.push(temp.leftChild);
               localStack.push(temp.rightChild);

               if(temp.leftChild != null ||
                                   temp.rightChild != null)
                  isRowEmpty = false;
               }
            else
               {
               System.out.print("--");
               localStack.push(null);
               localStack.push(null);
               }
            for(int j=0; j<nBlanks*2-2; j++)
               System.out.print(' ');
            }
            System.out.println();
            nBlanks /= 2;
            while(localStack.isEmpty()==false)
               globalStack.push( localStack.pop() );
            }
         System.out.println(
         "......................................................");
         }
// -------------------------------------------------------------
   }  // Конец класса Tree
////////////////////////////////////////////////////////////////
class TreeApp
   {
   public static void main(String[] args) throws IOException
      {
      int value;
      Tree theTree = new Tree();
      theTree.insert(50, 1.5);
      theTree.insert(25, 1.2);
      theTree.insert(75, 1.7);
      theTree.insert(12, 1.5);
      theTree.insert(37, 1.2);
      theTree.insert(43, 1.7);
      theTree.insert(30, 1.5);
```

продолжение ⇏

Листинг 8.1 (*продолжение*)

```
      theTree.insert(33, 1.2);
      theTree.insert(87, 1.7);
      theTree.insert(93, 1.5);
      theTree.insert(97, 1.5);

      while(true)
         {
         System.out.print("Enter first letter of show, ");
         System.out.print("insert, find, delete, or traverse: ");
         int choice = getChar();
         switch(choice)
            {
            case 's':
               theTree.displayTree();
               break;
            case 'i':
               System.out.print("Enter value to insert: ");
               value = getInt();
               theTree.insert(value, value + 0.9);
               break;
            case 'f':
               System.out.print("Enter value to find: ");
               value = getInt();
               Node found = theTree.find(value);
               if(found != null)
                  {
                  System.out.print("Found: ");
                  found.displayNode();
                  System.out.print("\n");
                  }
// -------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
// -------------------------------------------------------------
   public static char getChar() throws IOException
      {
      String s = getString();
      return s.charAt(0);
      }
//-------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
```

```
      }
// -------------------------------------------------------------
   }  // Конец класса TreeApp
////////////////////////////////////////////////////////////////
```

Для выхода из программы используется комбинация клавиш Ctrl+C; для простоты мы не стали программировать односимвольную команду для выполнения этой операции.

Код Хаффмана

Далеко не все двоичные деревья являются деревьями двоичного поиска. Многие двоичные деревья используются для других целей. Пример был приведен на рис. 8.11, где двоичное дерево использовалось для представления алгебраического выражения.

В этом разделе рассматривается алгоритм, использующий двоичное дерево с несколько неожиданной целью: для сжатия данных. Этот алгоритм называется кодом Хаффмана — по имени Дэвида Хаффмана, открывшего его в 1952 году. Сжатие данных играет важную роль во многих ситуациях — например, оно позволяет ускорить передачу данных по Интернету (особенно по модемному подключению). Реализация схемы получается довольно длинной, поэтому полный код программы в этом разделе не приводится. Мы ограничимся описанием основных концепций, а реализация предоставляется читателю для самостоятельной работы.

Коды символов

Каждый символ обычного несжатого текстового файла представляется одним байтом (традиционный код ASCII) или двумя байтами (более новая схема кодирования Юникод, которая проектировалась с целью поддержки любых языков). В этих схемах каждый символ кодируется одинаковым количеством битов. В таблице 8.3 приведены двоичные представления некоторых символов в коде ASCII. Как видно из таблицы, каждый символ представляется 8 битами.

Таблица 8.3. Некоторые ASCII-коды

Символ	Десятичное представление	Двоичное представление
A	65	01000000
B	66	01000001
C	67	01000001
...	...	...
X	88	01011000
Y	89	01011001
Z	90	01011010

Существуют разные способы сжатия данных. Для текста чаще всего применяется способ, основанный на уменьшении количества битов в представлении самых частых символов. В английском языке чаще всего встречается буква E, поэтому для ее кодирования логично выбрать как можно меньшее количество битов. С другой стороны, буква Z встречается редко, и большая длина ее кодировки уже не столь существенна.

Допустим, в представлении буквы E используются всего два бита, например 01. Закодировать двумя битами каждую букву алфавита не удастся, потому что существуют всего четыре двухразрядные комбинации: 00, 01, 10 и 11. Нельзя ли использовать эти комбинации для представления четырех наиболее часто используемых символов?

К сожалению, нельзя. Необходимо следить за тем, чтобы символы не начинались с комбинации битов, присутствующей в начале более длинного кода другого символа. Например, если буква E кодируется комбинацией 01, а буква X — комбинацией 01011000, то декодирующий алгоритм при обработке 01011000 не будет знать, является ли префикс 01 кодом буквы E или началом кода X. Отсюда следует правило:

Ни один код не должен быть префиксом другого кода.

Также необходимо учесть, что в некоторых сообщениях символ E может и не быть наиболее часто используемым символом. Например, если текст представляет собой файл с исходным кодом на языке Java, то символ «;» (точка с запятой) может встречаться чаще, чем E. Чтобы справиться с этой проблемой, мы будем строить для каждого сообщения новый код, адаптированный для этого конкретного сообщения. Допустим, потребовалось переслать сообщение «SUSIE SAYS IT IS EASY». Буква S встречается в нем чаще остальных букв — как и пробел. Создание кода начинается с построения *частотной таблицы*, в которой указана частота (количество вхождений) каждого символа (табл. 8.4).

Таблица 8.4. Частотная таблица

Символ	**Частота**
A	2
E	2
I	3
S	6
T	1
U	1
Y	2
Пробел	4
Символ новой строки	1

Символы с наибольшим количеством вхождений должны кодироваться малым количеством битов. В таблице 8.5 представлена одна из возможных кодировок приведенного ранее сообщения.

Таблица 8.5. Код Хаффмана

Символ	Код
A	010
E	1111
I	110
S	10
T	0110
U	01111
Y	1110
Пробел	00
Символ новой строки	01110

Буква S кодируется комбинацией 10, а пробел — комбинацией 00. Использовать комбинации 01 или 11 нельзя, потому что они являются префиксами других символов. Как насчет 3-разрядных комбинаций? Существуют четыре возможности: 000, 001, 010, 011, 100, 101, 110 и 111. Буква A кодируется комбинацией 010,а I — комбинацией 110. Почему не используются другие комбинации? Мы уже знаем, что в коде не могут использоваться комбинации, начинающиеся с 10 и 00; это исключает четыре возможности. Кроме того, 011 используется в начале U и символа новой строки, а 111 используется в начале E и Y. Остаются только два 3-разрядных кода, которые назначаются буквам A и I. Аналогичным образом можно понять, почему доступны только три 4-разрядных кода.

Таким образом, закодированное сообщение выглядит так:

```
10 01111 10 110 1111 00 10 010 1110 10 00 110 0110 00 110 10 00 1111 010 10 1110
01110
```

Для удобства чтения закодированное сообщение разбито на коды отдельных символов. Конечно, в действительности все биты идут сплошным потоком; двоичное сообщение не содержит пробелов, только двоичные 0 и 1.

Декодирование по дереву Хаффмана

О том, как строятся коды Хаффмана, будет рассказано позднее. Сначала давайте разберем более простой процесс декодирования. Предположим, вы получили цепочку битов, приведенную в предыдущем разделе. Как снова преобразовать ее в символы? Для этого можно воспользоваться двоичным деревом, называемым *деревом Хаффмана*. На рис. 8.23 изображено дерево Хаффмана для только что описанного кода.

Символам сообщения в дереве соответствуют листовые узлы. Чем чаще они встречаются в сообщении, тем выше в дереве находится соответствующий узел. Число рядом с кружком обозначает частоту вхождения. Рядом с не-листовыми узлами выводятся суммы частот потомков. Позднее мы увидим, для чего они нужны.

Как использовать это дерево для декодирования сообщения? Декодирование каждого символа начинается с корневого узла. Если в потоке обнаружен бит 0, вы переходите налево к следующему узлу, а если бит 1 — то направо. Попробуйте проделать это для кода A (010). Налево, направо, потом снова налево — и вы оказываетесь у узла A. Направления переходов обозначены стрелками на рис. 8.23.

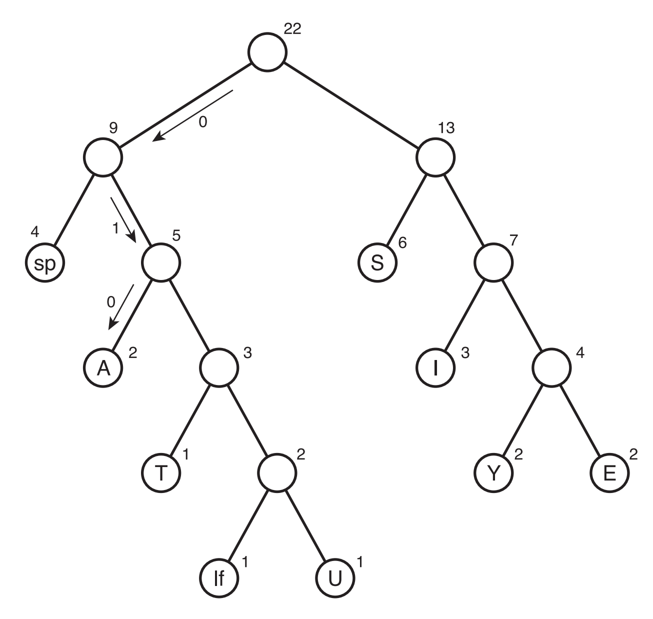


Рис. 8.23. Дерево Хаффмана

Другие символы декодируются аналогичным образом. Если вам хватит терпения, всю битовую строку можно декодировать подобным образом.

Построение дерева Хаффмана

Итак, как использовать дерево Хаффмана, понятно; но как это дерево построить? Существует много способов решения этой проблемы. Наш подход базируется на классах `Node` и `Tree` из программы `tree.java` в листинге 8.1 (хотя методы, относящиеся к деревьям поиска — `find()`, `insert()` и `delete()`, — перестают быть актуальными).

Алгоритм построения дерева выглядит так:

1. Создать объект `Node` (см. программу `tree.java`) для каждого символа, используемого в сообщении. В сообщении из нашего примера будет 9 узлов. Каждый узел состоит из двух элементов данных: символа и частоты этого символа в сообщении. Информация для сообщения «SUSIE SAYS IT IS EASY» приведена в табл. 8.4.
2. Создать объект дерева для каждого из этих узлов. Узел становится корнем дерева.

3. Вставить эти деревья в приоритетную очередь (см. главу 4). Деревья упорядочиваются по частоте, при этом наименьшая частота обладает наибольшим приоритетом. Таким образом, при извлечении всегда выбирается дерево с наименее часто используемым символом.

Далее происходит следующее:

1. Извлечь два дерева из приоритетной очереди и сделать их потомками нового узла. Частота нового узла является суммой частот потомков; поле символа может остаться пустым.
2. Вставить новое дерево из трех узлов обратно в приоритетную очередь.
3. Продолжить выполнение шагов 1 и 2. Деревья постепенно увеличиваются, а их количество постепенно сокращается. Когда в очереди останется только одно дерево, оно представляет собой дерево Хаффмана. Работа алгоритма на этом завершается.

На рис. 8.24 и 8.25 показано, как строится дерево Хаффмана для сообщения «SUSIE SAYS IT IS EASY».

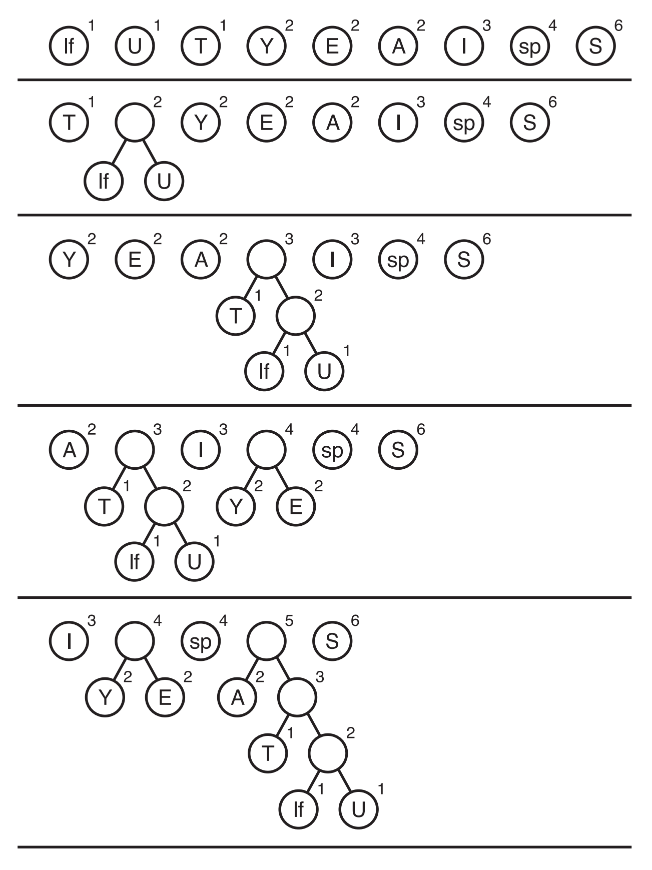


Рис. 8.24. Построение дерева Хаффмана, часть 1

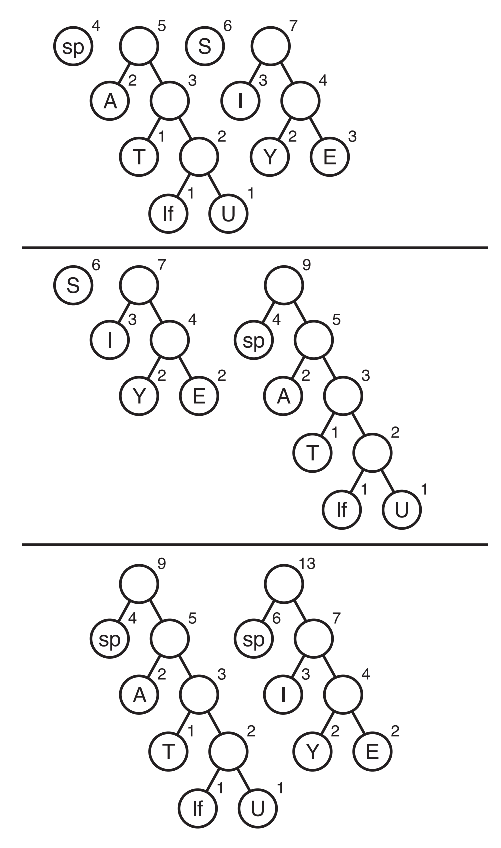


Рис. 8.25. Построение дерева Хаффмана, часть 2

Кодирование сообщения

Итак, у нас имеется дерево Хаффмана; как закодировать сообщение? Прежде всего необходимо создать кодовую таблицу, в которой для каждого символа приводится соответствующий код Хаффмана. Чтобы упростить описание, будем считать, что вместо кода ASCII наш компьютер использует упрощенный алфавит из 28 символов верхнего регистра. Буква A обозначается кодом 0, B — 1 и так далее до буквы Z, которая обозначается кодом 25. Пробелу ставится в соответствие код 26, а символу новой строки — код 27. (Конечно, это не код со сжатием, а всего лишь упрощение кода ASCII — обычного способа хранения символьной информации в компьютере.)

Наша кодовая таблица будет представлять собой массив из 28 ячеек. Индекс каждой ячейки соответствует числовому значению символа: 0 для A, 1 для B и т. д. В ячейке хранится код Хаффмана соответствующего символа. Данные содержатся не во всех ячейках, а только в ячейках символов, присутствующих в сообщении. На рис. 8.26 показано, как выглядит этот массив для сообщения «SUSIE SAYS IT IS EASY».

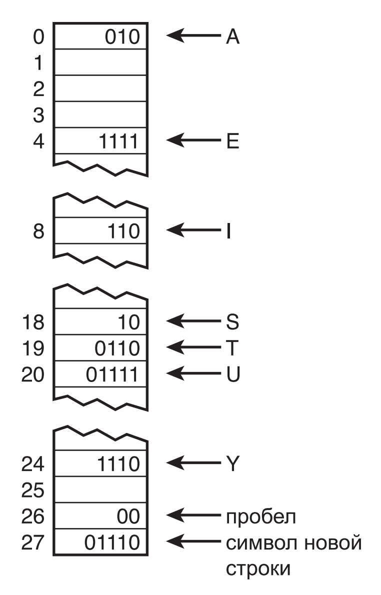


Рис. 8.26. Построение дерева Хаффмана, часть 3

Кодовая таблица упрощает генерирование кодового сообщения: для каждого символа из исходного сообщения его код используется в качестве индекса кодовой таблицы. Далее коды Хаффмана раз за разом присоединяются к кодированному сообщению, пока оно не будет завершено.

Создание кода Хаффмана

Как создать код Хаффмана, хранимый в кодовой таблице? Процесс напоминает декодирование сообщения. Начните с корня дерева Хаффмана и отследите все возможные пути к листовым узлам. В процессе перемещения сохраняется выбор направления на каждой «развилке»: 0 для левого ребра, 1 для правого ребра. Когда вы приходите к листовому узлу символа, последовательность 0 и 1 образует код Хаффмана для этого символа. Код заносится в соответствующую ячейку кодовой таблицы.

Построение кода может быть реализовано посредством вызова метода, который начинается от корня таблицы, а затем рекурсивно вызывает себя для каждого потомка. Через некоторое время алгоритм обойдет все пути ко всем листовым узлам, и кодовая таблица будет построена.

Итоги

- ✦ Деревья состоят из узлов, соединенных ребрами.
- ✦ Корневым узлом (или корнем) называется узел верхнего уровня дерева. Корневой узел не имеет родителя.

- В двоичном дереве узел имеет не более двух потомков.
- В дереве двоичного поиска ключи всех узлов, являющиеся левыми потомками узла A, меньше ключа A; ключи всех узлов, являющиеся правыми потомками узла A, больше ключа A (или равны ему).
- Поиск, вставка и удаление в деревьях выполняются за время $O(\log N)$.
- Узлы соответствуют объектам данных, хранимым в дереве.
- Ребра обычно представляются в программах ссылками на потомков узла (а иногда и на родителя).
- Обходом дерева называется посещение всех его узлов в определенном порядке.
- Простейшие алгоритмы обхода — прямой, симметричный и обратный.
- *Несбалансированным* называется дерево, у которого корень имеет больше левых потомков (во всех поколениях), чем правых (или наоборот).
- В процессе поиска узла алгоритм сравнивает искомое значение с ключом узла и переходит к левому потомку этого узла, если искомое значение меньше, или к правому, если оно больше.
- При выполнении вставки алгоритм сначала находит место для нового узла, а затем изменяет ссылку на потомка в родительском узле, чтобы она указывала на вставленный узел.
- При *симметричном* обходе узлы посещаются в порядке возрастания ключей.
- *Прямой* и *обратный* обход полезны при разборе алгебраических выражений.
- Если узел не имеет потомков, то для его удаления достаточно записать `null` в поле ссылки на потомка в его родительском узле.
- Чтобы удалить узел с одним потомком, следует записать ссылку на его потомка в поле родительского узла.
- Чтобы удалить узел с двумя потомками, следует заменить его преемником.
- Преемник узла A находится поиском наименьшего узла в поддереве, корнем которого является правый потомок A.
- При удалении узла с двумя потомками возможны разные ситуации в зависимости от того, является ли преемник правым потомком удаляемого узла или одним из левых потомков (во всех поколениях) правого потомка.
- Узлы с совпадающими значениями ключей могут создать проблемы в массивах, потому что при поиске будет найден только первый из них.
- Деревья могут представляться в памяти компьютера в виде массива, хотя представление со ссылками является более распространенным.
- Деревом Хаффмана называется двоичное дерево (но не дерево двоичного поиска!), использующее алгоритм сжатия данных, называемый *кодом Хаффмана*.
- В коде Хаффмана часто встречающиеся символы кодируются наименьшим, а самые редкие символы — наибольшим количеством битов.

Вопросы

Следующие вопросы помогут читателю проверить качество усвоения материала. Ответы на них приведены в приложении В.

1. За какое время (в O-синтаксисе) выполняются операции вставки и удаления в дереве?
2. Двоичное дерево называется деревом поиска, если:
   a) у каждого нелистового узла имеются потомки, ключи которых меньше (либо равны) ключа родителя;
   b) ключ каждого левого потомка меньше, чем у родителя, а ключ каждого правого потомка больше ключа родителя (либо равен ему);
   c) на пути от корня к каждому листовому узлу ключ каждого узла больше (либо равен) ключа его родителя;
   d) узел может иметь не более двух потомков.
3. Не все деревья являются двоичными деревьями (Да/Нет).
4. Имеется полное двоичное дерево с 20 узлами. Предполагается, что корень находится на уровне 0. Сколько узлов находится на уровне 4?
5. У поддерева двоичного дерева всегда:
   a) корень является потомком корня основного дерева;
   b) корень не связан с корнем основного дерева;
   c) количество узлов меньше, чем в основном дереве;
   d) существует одноранговый узел с таким же количеством узлов.
6. В реализации дерева на языке Java _______ и _______ обычно представляются разными классами.
7. Алгоритм поиска в дереве двоичного поиска перемещается от узла к узлу и проверяет:
   a) в каком отношении (больше/меньше) ключ текущего узла находится с искомым ключом;
   b) в каком отношении (больше/меньше) ключ текущего узла находится с его правым или левым потомком;
   c) не является ли текущий листовой узел искомым;
   d) на каком уровне находится текущая позиция поиска.
8. Несбалансированным называется дерево:
   a) в котором значения большинство ключей больше среднего арифметического;
   b) обладающее непредсказуемым поведением;
   c) в котором количество левых потомков у корневого или другого узла значительно превышает количество правых потомков (или наоборот);
   d) имеющее форму зонтика.

9. Вставка узла начинается с тех же действий, что и ________ узла.
10. Допустим, преемником узла A является узел S. В этом случае ключ узла S должен быть больше ______, но меньше либо равен ________.
11. Какое из следующих утверждений *ложно* по отношению к дереву, используемому для представления математических выражений?
    a) Оба потомка узла оператора должны быть операндами.
    b) После обхода в обратном порядке не нужно добавлять круглые скобки.
    c) После обхода в симметричном порядке необходимо добавлять круглые скобки.
    d) При обходе в прямом порядке узел посещается ранее любого из его потомков.
12. Если для представления дерева используется массив, то правый потомок узла с индексом *n* имеет индекс ________ .
13. Чтобы удалить узел с одним потомком из дерева двоичного поиска, необходимо найти преемника этого узла (Да/Нет).
14. Дерево Хаффмана обычно используется для ________ текста.
15. Какое из следующих утверждений *ложно* по отношению к дереву Хаффмана?
    a) Наиболее часто используемые символы всегда находятся ближе к вершине дерева.
    b) При декодировании сообщения обычно приходится многократно проходить от корня к листовому узлу.
    c) При кодировании символа алгоритм обычно начинается с листового узла и двигается наверх.
    d) Для генерирования дерева могут использоваться операции извлечения и вставки в приоритетной очереди.

Упражнения

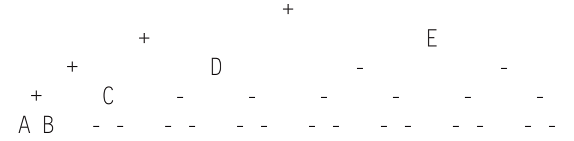
Упражнения помогут вам глубже усвоить материал этой главы. Программирования они не требуют.

1. Создайте 20 деревьев в приложении Binary Tree Workshop. Какой примерно процент среди них составляют серьезно несбалансированные деревья?
2. Постройте диаграмму UML (или блок-схему — для ветеранов от программирования) для различных ситуаций, возможных при удалении узла из дерева двоичного поиска. На диаграмме должны быть представлены все три случая, описанных в тексте главы. Включите действия для левых и правых потомков, а также особые случаи вроде удаления корневого узла. Например, в случае 1 возможны два варианта (левый и правый потомки). Блоки в конце каждого пути должны содержать описание того, как выполняется удаление в этой ситуации.
3. Используйте приложение Binary Tree Workshop для удаления узла во всех возможных ситуациях.

Программные проекты

В этом разделе читателю предлагаются программные проекты, которые укрепляют понимание материала и демонстрируют практическое применение концепций этой главы.

8.1. Измените программу `tree.java` (см. листинг 8.1) так, чтобы она создавала двоичное дерево по вводимой пользователем цепочке символов (например, A, B и т. д.). Каждая буква отображается в собственном узле. Дерево должно строиться так, чтобы все узлы, содержащие буквы, были листовыми. Родительские узлы могут содержать какой-нибудь символ, не являющийся буквой, например +. Проследите за тем, чтобы каждый родительский узел имел ровно двух потомков. Неважно, если дерево получится несбалансированным. Обратите внимание: созданное дерево не является деревом поиска; в нем не существует быстрого способа найти заданный узел. Дерево, которое у вас получится, выглядит именно так:



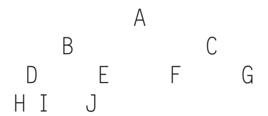
```
                                +
            +                               E
      +             D               -               -
   +     C      -      -      -      -      -      -
A B   - -   - -   - -   - -   - -   - -   - -
```

Например, можно начать с создания массива деревьев. (Группа несвязанных деревьев называется *лесом*.) Каждая буква, вводимая пользователем, сохраняется в узле. Узел помещается в дерево, корнем которого он становится. Далее все эти деревья, состоящие из одного узла, помещаются в массив. Начните с создания нового дерева с корнем + и двумя деревьями из одного узла в качестве потомков. Затем продолжайте включать в это дерево одноузловые деревья из массива. Не беспокойтесь, если дерево получится несбалансированным. Промежуточное дерево можно сохранить в массиве поверх дерева, содержимое которого уже было добавлено ранее.

Методы `find()`, `insert()` и `delete()`, относящиеся только к деревьям поиска, можно удалить. Оставьте метод `displayTree()` и методы обхода, потому что они работают с любыми двоичными деревьями.

8.2. Усовершенствуйте программу из п. 8.1, чтобы она создавала сбалансированное дерево. Одно из возможных решений — проследить за тем, чтобы на нижнем уровне было как можно больше листовых узлов. Для начала создайте трехузловое дерево из каждой пары одноузловых деревьев, с новым корневым узлом +. В результате создается лес трехузловых деревьев. Затем каждая пара трехузловых деревьев объединяется для создания леса семиузловых деревьев. С ростом количества узлов в каждом дереве количество деревьев уменьшается, пока не останется только одно дерево.

8.3. Снова начните с программы `tree.java` и постройте дерево из символов, вводимых пользователем. На этот раз должно создаваться *полное* дерево, то есть дерево, содержащее все возможные узлы на всех уровнях (кроме правого края последнего ряда). Символы должны быть упорядочены сверху вниз и слева направо в каждом ряду, как для текста, написанного на пирамиде. (Такое расположение не соответствует ни одному из трех порядков обхода, рассматривавшихся в этой главе.) Таким образом, строка ABCDEFGHIJ будет упорядочена в виде



Например, такое дерево можно строить сверху вниз (а не снизу вверх, как в двух предыдущих программных проектах). Начните с создания узла, который станет корнем итогового дерева. Если представить, что узлы нумеруются в порядке следования букв (корневому узлу присваивается номер 1), то у любого узла с номером n левый потомок имеет номер $2 * n$, а правый — $2 * n + 1$. В решении можно использовать рекурсивный метод, который создает двух потомков, а затем вызывает себя для каждого потомка. Узлы не обязательно создавать в порядке их размещения на дереве. Как и в предыдущих проектах, функции для работы с деревом поиска можно удалить из класса `Tree`.

8.4. Напишите программу, которая преобразует постфиксное выражение в дерево, изображенное на рис. 8.11 этой главы. Вам придется внести изменения в класс `Tree` из программы `tree.java` (см. листинг 8.1) и класс `ParsePost` из программы `postfix.java` (см. листинг 4.8) главы 4. Более подробная информация приведена в комментариях к рис. 8.11.

После того, как дерево будет сгенерировано, обход дерева позволит получить префиксный, инфиксный и постфиксный эквиваленты алгебраического выражения. В инфиксной версии потребуются круглые скобки для предотвращения неоднозначности генерируемых выражений. В методе `inOrder()` перед первым рекурсивным вызовом выводится открывающая круглая скобка, а после второго — закрывающая.

8.5. Напишите программную реализацию кодирования и декодирования Хаффмана. Программа должна делать следующее:

- ✦ Получать текстовое сообщение — возможно, состоящее из нескольких строк.
- ✦ Создавать дерево Хаффмана для этого сообщения.
- ✦ Создавать кодовую таблицу.
- ✦ Кодировать сообщение в двоичную форму.
- ✦ Декодировать сообщение из двоичной формы обратно в текстовую.

Для коротких сообщений программа должна выводить построенное дерево Хаффмана. Вероятно, в работе над проектом вам помогут описания из п. 8.1, 8.2 и 8.3. Для хранения последовательностей двоичных нулей и единиц можно воспользоваться переменными `String`. Не используйте поразрядные операции с битами без необходимости.

Глава 9
Красно-черные деревья

Как вы узнали в главе 8, «Двоичные деревья», у обычных деревьев двоичного поиска имеются важные преимущества, которые делают их ценным средством хранения данных: возможность быстрого поиска элемента с заданным ключом, а также быстрое выполнение операций вставки и удаления элементов. Другие структуры данных — массивы, сортированные массивы, связанные списки — по крайней мере в одной из этих областей работают недостаточно быстро. На первый взгляд может показаться, что деревья двоичного поиска являются идеальной структурой данных.

К сожалению, у обычных деревьев двоичного поиска имеется одна неприятная особенность. Они хорошо работают, если данные вставляются в дерево в произвольном порядке. Но если вставляемые данные изначально отсортированы в прямом (17, 21, 28, 36,...) или обратном порядке (36, 28, 21, 17,...), эффективность деревьев резко снижается. Если вставляемые значения уже упорядочены, двоичное дерево становится несбалансированным. В несбалансированном дереве теряется возможность быстрого поиска (а также вставки или удаления).

В этой главе рассматривается одно из решений проблемы несбалансированности деревьев: красно-черные деревья представляют собой деревья двоичного поиска с рядом дополнительных возможностей.

Существуют и другие способы обеспечения сбалансированности деревьев. Одни из них упоминаются ближе к концу этой главы, а другие — деревья 2-3-4 и деревья 2-3 — рассматриваются в главе 10, «Деревья 2-3-4». Более того, как будет показано в этой главе, операции с деревом 2-3-4 удивительным образом соответствуют операциям с красно-черными деревьями.

Наш подход к изложению темы

Операция вставки в красно-черных деревьях будет объясняться несколько иначе, чем вставка в других структурах данных. Красно-черные деревья не назовешь простыми для понимания. По этой причине, а также из-за обилия симметричных случаев (левые или правые потомки, внутренние или внешние внуки и т. д.) код реализации получается более длинным и сложным, чем можно было бы предположить. Изучать алгоритмы красно-черных деревьев на примере кода довольно сложно, поэтому в этой главе нет ни одного листинга. Аналогичную функциональность можно реализовать для деревьев 2-3-4 при помощи кода, приведенного в главе 10. Тем не менее концепции, изложенные в этой главе, помогут вам понять деревья 2-3-4, которые сами по себе довольно интересны.

Концептуальное понимание

В понимании красно-черных деревьев на концептуальном уровне нам поможет приложение RBTree Workshop. Вы узнаете, как использовать это приложение для вставки новых узлов в дерево. Конечно, ваше участие в процедуре вставки замедляет ее, но также поможет вам понять, как это происходит.

Поиск в красно-черных деревьях работает так же, как в обычных двоичных деревьях. С другой стороны, операции вставки и удаления, хотя и базируются на алгоритмах обычных деревьев, но подвергаются значительным модификациям. Соответственно в этой главе основное внимание будет уделяться процессу вставки.

Нисходящая вставка

Способ вставки, который мы будем рассматривать, называется *нисходящей* (top-down) вставкой. В этом способе при спуске алгоритма вниз по дереву в поисках места для вставки узла в дерево могут вноситься структурные изменения.

Другой способ вставки называется *восходящей* вставкой. Алгоритм сначала находит место для вставки узла, а потом возвращается вверх по дереву, внося структурные изменения. Восходящая вставка менее эффективна, потому что она требует двух проходов по дереву.

Сбалансированные и несбалансированные деревья

Прежде чем браться за анализ красно-черных деревьев, давайте сначала разберемся, как деревья становятся несбалансированными. Запустите приложение Binary Tree Workshop из главы 8 (*не* приложение RBTree этой главы!). Воспользуйтесь кнопкой `Fill` для создания дерева, содержащего всего один узел. Затем вставьте серию узлов, ключи которых следуют в порядке возрастания или убывания. Результат будет выглядеть примерно так, как показано на рис. 9.1.

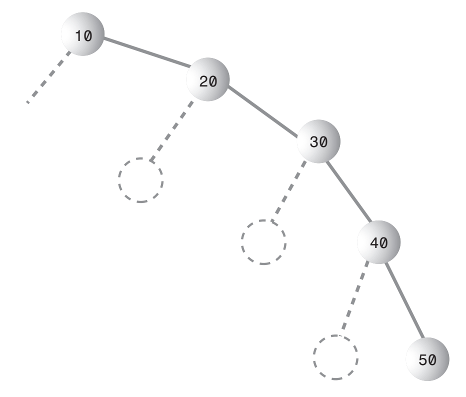


Рис. 9.1. Вставка элементов, упорядоченных по возрастанию

Узлы выстраиваются в линию без ветвления. Поскольку каждый узел больше узла, вставленного перед ним, каждый узел является правым потомком, поэтому все узлы располагаются по одну сторону от корня. Дерево получается предельно несбалансированным. Если вставить элементы, упорядоченные по убыванию, то каждый узел будет левым потомком своего родителя, а дерево окажется несбалансированным с другой стороны.

Вырождение до O(N)

При отсутствии ветвей дерево фактически превращается в связанный список. Двумерная структура данных становится одномерной. К сожалению, как и в случае со связанным списком, для поиска нужного элемента теперь придется просмотреть (в среднем) половину элементов. В такой ситуации скорость поиска сокращается до $O(N)$ — вместо $O(\log N)$, как у сбалансированного дерева. Поиск среди 10 000 элементов в таком несбалансированном дереве потребует в среднем 5000 сравнений, тогда как в сбалансированном дереве со случайными вставками хватит всего 14. Для предварительно отсортированных данных с таким же успехом можно было изначально воспользоваться связанным списком.

На основе частично отсортированных данных строятся деревья, несбалансированные только частично. Воспользуйтесь приложением Binary Tree Workshop из главы 8 и сгенерируйте несколько деревьев с 31 узлом — вы увидите, что некоторые из этих деревьев являются более несбалансированными, чем другие (рис. 9.2).

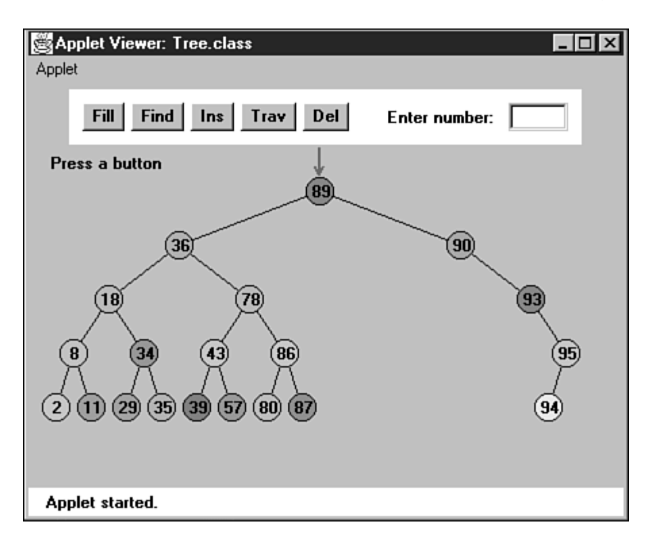


Рис. 9.2. Частично несбалансированное дерево

Хотя эффективность такого дерева все же лучше, чем у максимально несбалансированного дерева, время поиска в нем не оптимально.

В приложении Binary Tree Workshop деревья иногда становятся частично несбалансированными даже при случайно сгенерированных данных, потому что при

столь малом объеме данных даже короткая серия упорядоченных чисел окажет заметное влияние на дерево. Кроме того, балансировку дерева могут нарушить очень маленькие или очень большие значения ключей, фактически блокирующие возможность вставки с той или иной стороны от узла. Например, слева от корневого узла с ключом 3 возможна вставка только двух узлов.

При более или менее реалистичных объемах случайных данных дерево вряд ли подвергнется серьезной разбалансировке. Тем не менее серии отсортированных данных приведут к частичному нарушению сбалансированности дерева. У частично несбалансированных деревьев время поиска лежит где-то в диапазоне между $O(N)$ и $O(\log N)$ в зависимости от степени разбалансированности.

Спасительный баланс

Чтобы обеспечить быстрое время поиска $O(\log N)$, на которое способно дерево, необходимо позаботиться о том, чтобы дерево всегда оставалось сбалансированным (или по крайней мере почти сбалансированным). Это означает, что каждый узел дерева должен иметь справа и слева приблизительно одинаковое количество потомков (во всех поколениях).

В красно-черном дереве баланс обеспечивается при вставке (а также при удалении, но мы пока на это не будем обращать внимания). Алгоритм следит за тем, чтобы некоторые характеристики дерева не нарушались в результате вставки. В случае нарушения алгоритм принимает меры и изменяет структуру дерева по мере необходимости. Сохранение этих характеристик обеспечивает сбалансированность дерева.

Характеристики красно-черного дерева

Что же это за загадочные характеристики дерева? Их две — одна простая, а другая посложнее:

- Каждый узел окрашен в определенный цвет.
- В процессе вставки и удаления выполняются различные правила взаимного расположения этих цветов.

Цветные узлы

В красно-черном дереве каждый узел окрашен либо в красный, либо в черный цвет. Цвета выбраны произвольно; например, их можно было бы заменить синим и желтым. Более того, сама концепция «цветов» выбрана произвольно: с таким же успехом можно было разделить узлы на тяжелые или легкие, или на узлы-*ин* и узлы-*янь*. Тем не менее цвета достаточно удобны в качестве меток. Для представления информации о цвете в класс узла включается поле данных, которое может относиться к логическому типу (например, `isRed`).

В приложении RBTree Workshop цвет узла обозначается цветом его контура. Цвет заливки, как и в приложении Binary Tree Workshop из предыдущей главы, генерируется случайным образом.

Говоря о цвете узла в этой главе, мы почти всегда имеем в виду красный или черный контур. На иллюстрациях (кроме снимка экрана на рис. 9.3) черные узлы будут обозначаться сплошным черным контуром, а красные узлы — белым контуром. (Некоторые узлы изображаются вообще без контура — это означает, что цвет узла в данном случае роли не играет.)

Красно-черные правила

При вставке (или удалении) узла должны выполняться некоторые правила, которые мы будем называть *красно-черными правилами*. Если эти правила выполняются, дерево остается сбалансированным. Краткий перечень этих правил:

1. Каждый узел окрашен в красный или черный цвет.
2. Корень всегда окрашен в черный цвет.
3. Если узел красный, то его потомки должны быть черными (хотя обратное не всегда истинно).
4. Все пути от корня к узлу или пустому потомку должны содержать одинаковое количество черных узлов.

Пустым потомком, упомянутым в правиле 4, называется место возможного присоединения потомка к не-листовому узлу. Другими словами, это отсутствующий левый потомок узла, имеющего правого потомка, или отсутствующий правый потомок узла с левым потомком. Не беспокойтесь, скоро все станет более понятным.

Количество черных узлов на пути от корня к листу называется *черной высотой*. В другой формулировке правило 4 может гласить, что все пути от корня к узлу должны иметь одинаковую черную высоту.

Пока правила выглядят абсолютно загадочно. Совершенно неясно, почему они должны приводить к построению сбалансированного дерева, и все же это как-то происходит. Запишите их на бумажке и держите под рукой. Вам довольно часто придется обращаться к ним в ходе чтения этой главы.

Приложение RBTree Workshop поможет лучше понять, как работают эти правила. Вскоре мы поэкспериментируем с приложением, но сначала необходимо понять, что делать в случае нарушения одного из красно-черных правил.

Дубликаты ключей

Что произойдет, если дерево содержит несколько элементов с одинаковыми ключами? Здесь в красно-черных деревьях возникает небольшая проблема. Очень важно, чтобы узлы с одинаковыми ключами были равномерно распределены по обе стороны от других узлов с тем же ключом. Иначе говоря, если ключи поступают в порядке 50, 50, 50, то второй элемент 50 должен располагаться справа от первого, а третий — слева от первого. В противном случае дерево перестанет быть сбалансированным.

Распределение узлов с одинаковыми ключами может быть выполнено посредством введения некоторого процесса рандомизации в алгоритм вставки. Тем не менее это приведет к усложнению процесса поиска, если потребуется найти все элементы с одинаковыми ключами.

Проще запретить существование элементов с одинаковыми ключами. В следующем описании предполагается, что дубликаты в дереве запрещены.

Исправление нарушений

Предположим, вы видите (или приложение сообщает вам) о нарушении красно-черных правил. Как исправить ситуацию и привести дерево к виду, удовлетворяющему правилам? Возможны два (и только два!) действия:

- Изменение цвета узлов.
- Выполнение поворотов.

В приложении под изменением цвета узла понимается изменение цвета его контура (а не заливки). *Поворотом* называется такая перегруппировка узлов, в результате которой дерево становится более сбалансированным.

Вероятно, сейчас вам эти концепции кажутся предельно абстрактными и невразумительными. Приложение RBTree Workshop поможет прояснить ситуацию.

Работа с приложением RBTree Workshop

На рис. 9.3 показано, как выглядит приложение RBTree Workshop после вставки нескольких узлов. (На рисунке отличить черный контур от красного довольно трудно, но на цветном мониторе все просто.)

Приложение RBTree содержит довольно много кнопок. Мы кратко разберемся с тем, что они делают, хотя на этой стадии некоторые описания выглядят довольно загадочно.

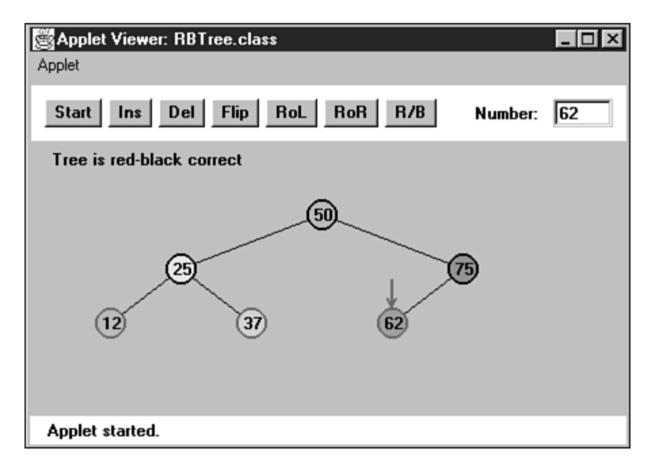


Рис. 9.3. Приложение RBTree Workshop

Щелчок на узле

Красная стрелка указывает на текущий выделенный узел. Это тот самый узел, который перекрашивается в другой цвет или который является *верхним узлом* при повороте. Чтобы выделить узел, щелкните на нем кнопкой мыши; красная стрелка перемещается к выбранному узлу.

Кнопка Start

При первом запуске приложения RBTree Workshop (а также при нажатии кнопки `Start`) создается дерево только с одним узлом. Так как понимание красно-черных деревьев зависит от их использования в процессе вставки, удобнее начать с корневого узла и строить дерево посредством вставки дополнительных узлов. Для упрощения будущих операций корневому узлу всегда присваивается значение 50. Числа для последующих вставок вы выбираете самостоятельно.

Кнопка Ins

Кнопка `Ins` создает новый узел с ключом, введенным в поле `Number`, и вставляет его в дерево. (По крайней мере это происходит, если не требуется переключение цветов. За подробностями обращайтесь к описанию кнопки `Flip`.)

Обратите внимание: кнопка `Ins` выполняет всю операцию вставки за одно нажатие; в отличие от приложения Binary Tree Workshop из предыдущей главы, многократные нажатия не требуются. Следовательно, значение ключа должно быть введено перед нажатием кнопки. В приложении RBTree нас интересует не поиск места для вставки узла — он мало отличается от аналогичной процедуры в обычном дереве двоичного поиска, а сохранение сбалансированности дерева. Соответственно приложение не выводит информацию об отдельных действиях, выполняемых в процессе вставки.

Кнопка Del

Кнопка `Del` удаляет узел с ключом, введенным в поле `Number`. Как и в случае с кнопкой `Ins`, удаление выполняется сразу же после первого нажатия; повторные нажатия не нужны.

Кнопки `Del` и `Ins` используют базовые алгоритмы вставки — такие же, как в приложении Tree Workshop. Так организовано «разделение обязанностей» между приложением и пользователем: приложение выполняет вставку, но пользователь (большей частью) обязан принять меры к тому, чтобы красно-черные правила выполнялись, а дерево, соответственно, оставалось сбалансированным.

Кнопка Flip

Если красная стрелка установлена на черном родителе с двумя красными потомками (чтобы установить ее, щелкните на узле кнопкой мыши), то при последующем нажатии кнопки Flip родитель станет красным, а потомки — черными. Иначе говоря, происходит переключение цветов родителя и детей. Позднее вы узнаете, для чего может потребоваться подобная трансформация.

При попытке переключения цвета корневого узла он сам остается черным (согласно правилу 2), но цвет его потомков изменяется с красного на черный.

Кнопка RoL

Кнопка RoL выполняет левый поворот. Чтобы повернуть группу узлов, сначала установите стрелку на верхнем узле группы. Для выполнения левого поворота верхний узел должен иметь правого потомка. Далее остается лишь нажать кнопку RoL. Повороты более подробно рассматриваются ниже.

Кнопка RoR

Кнопка RoR выполняет правый поворот. Установите стрелку на верхнем узле, участвующем в повороте, убедитесь в том, что у него имеется левый потомок, и нажмите кнопку.

Кнопка R/B

Кнопка R/B перекрашивает красный узел в черный цвет, или наоборот. Щелкните кнопкой мыши, чтобы подвести красную стрелку к узлу, и нажмите кнопку. (Кнопка R/B изменяет цвет только одного узла; не путайте ее с кнопкой Flip, которая изменяет цвет сразу трех узлов.)

Текстовые сообщения

Сообщения в текстовом поле под кнопками указывают, является ли дерево правильным красно-черным деревом. Дерево считается правильным красно-черным деревом, если оно выполняет приведенные ранее правила с 1 по 4. Если дерево неправильное, сообщения подскажут, какое из правил нарушено. В некоторых случаях красная стрелка будет указывать на место нарушения.

Где кнопка Find?

В красно-черных деревьях поиск работает точно так же, как в обычных деревьях двоичного поиска из предыдущей главы. Алгоритм начинает перемещение от корня дерева, а затем на каждом встреченном узле (текущий узел) решает, перейти

ли к левому или правому потомку, сравнивая ключ текущего узла с искомым ключом.

Кнопка Find в приложении RBTree отсутствует, потому что вы уже понимаете суть процесса поиска, а нас сейчас интересует изменение красно-черных аспектов дерева.

Эксперименты с приложением Workshop

Теперь, когда вы представляете себе, что делают кнопки приложения RBTree, мы проведем несколько простых экспериментов. Они наглядно покажут, как работает приложение. Пока ваша задача заключается в освоении средств управления приложением. Позднее эти навыки будут использованы для балансировки дерева.

Эксперимент 1. Вставка двух красных узлов

Нажмите кнопку Start, чтобы удалить все лишние узлы. В дереве остается только корневой узел, которому всегда присваивается значение 50.

Вставьте новый узел со значением меньшим, чем у корневого узла (скажем, 25), введите число в поле Number и нажмите кнопку Ins. Включение этого узла не нарушает никаких правил, поэтому сообщение по-прежнему гласит, что дерево остается правильным красно-черным деревом.

Вставьте второй узел со значением большим, чем у корневого узла, например 75. Дерево остается правильным, а также сбалансированным — справа от единственного не-листового узла (корня) расположено столько же узлов, как и слева. Результат показан на рис. 9.4.

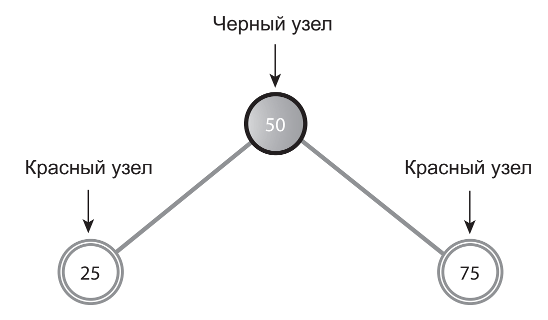


Рис. 9.4. Сбалансированное дерево

Только что вставленные узлы всегда окрашиваются в красный цвет (кроме корня). Выбор цвета не случаен — вставка красного узла с меньшей вероятностью нарушит красно-черные правила, чем вставка черного узла. Дело в том, что если новый красный узел присоединяется к черному узлу, никакие правила не нарушаются. Такая вставка не создает ситуации, в которой два красных узла следуют подряд (правило 3), и не изменяет черной высоты каких-либо путей (правило 4),

Конечно, если новый красный узел присоединяется к красному узлу, правило 3 будет нарушено, однако статистически это происходит только в половине случаев. Кроме того, добавление нового черного узла всегда приведен к изменению черной высоты его пути, а следовательно, к нарушению правила 4.

Кроме того, как будет показано далее, нарушения правила 3 (и родитель, и потомок являются красными узлами) исправляются проще, чем нарушения правила 4 (разные черные высоты).

Эксперимент 2. Повороты

Попробуем выполнить повороты. Начнем с трех узлов, показанных на рис. 9.4. Установите красную стрелку над корневым узлом (50), щелкнув на нем кнопкой мыши; этот узел будет верхним узлом поворота. Теперь выполните правый поворот нажатием кнопки RoR. Все узлы перемещаются в новые позиции (рис. 9.5).

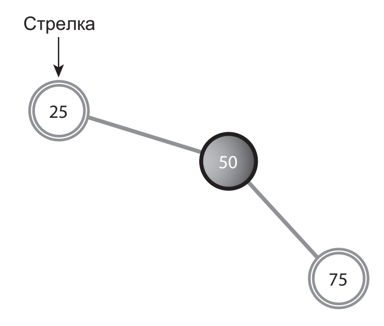


Рис. 9.5. После выполнения правого поворота

В результате правого поворота родитель занимает место своего правого потомка, левый потомок перемещается вверх и занимает место родителя, а правый потомок сдвигается вниз и становится «внуком» нового верхнего узла.

Обратите внимание: в результате операции дерево стало разбалансированным; справа от корня находится больше узлов, чем слева. Кроме того, сообщение указывает на нарушение красно-черных правил, а именно правила 2 (корень всегда окрашен в черный цвет). Пока не беспокойтесь об этой проблеме, а выполните поворот в другую сторону. Установите стрелку на узле 25, который теперь стал корневым (стрелка должна уже указывать на узел 25 после предыдущего поворота), и нажмите кнопку RoL. Узлы возвращаются к конфигурации на рис. 9.4.

Эксперимент 3. Переключение цветов

Начните с конфигурации на рис. 9.4, в которой после корневого узла 50 были вставлены узлы 25 и 75. Обратите внимание на то, что родитель (корень) является черным узлом, а оба его потомка — красными. Теперь попробуйте вставить другой узел. Какое бы значение ни было выбрано, выводится сообщение о невозможности вставки и необходимости переключения цветов.

Как упоминалось ранее, переключение цветов необходимо в том случае, когда в результате вставки создается черный узел с двумя красными потомками.

Красная стрелка должна быть уже установлена на черном родителе (корневом узле); нажмите кнопку `Flip`. Два потомка корневого узла окрашиваются из красного цвета в черный. Обычно родитель тоже меняет цвет и превращается из черного узла в красный, но сейчас мы имеем дело с особым случаем: корневой узел остается черным, чтобы избежать нарушения правила 2. Теперь все три узла окрашены в черный цвет, а дерево остается правильным красно-черным деревом.

Снова вставьте новый узел при помощи кнопки `Ins`. На рис. 9.6 показано, как выглядит дерево после вставки нового узла с ключом 12.

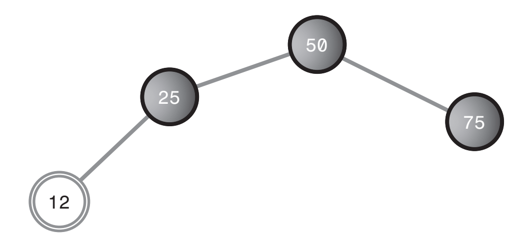


Рис. 9.6. После переключения цветов и вставки нового узла

Дерево по-прежнему остается правильным красно-черным деревом. Его корень окрашен в черный цвет, в дереве нет ситуаций, когда родитель и потомок одновременно были бы красными, а все пути имеют одинаковое количество черных узлов (два). Добавление нового красного узла не изменило красно-черной правильности дерева.

Эксперимент 4. Несбалансированное дерево

Теперь посмотрим, что произойдет при выполнении операции, нарушающей балансировку дерева. На рис. 9.6 один из путей содержит на один узел больше другого. Это нельзя назвать серьезной разбалансировкой, красно-черные правила не нарушаются, поэтому красно-черные алгоритмы считают, что причин для беспокойства нет. Но что, если один путь будет отличаться от другого на два и более уровня (под уровнем понимается количество узлов в пути)? В таких ситуациях красно-черные правила заведомо нарушаются, и алгоритму придется восстанавливать сбалансированность дерева.

Вставьте узел 6 в дерево на рис. 9.6. Появляется сообщение об ошибке: родитель и потомок одновременно являются красными узлами. Нарушается правило 3 (рис. 9.7).

Как исправить дерево, чтобы правило 3 не нарушалось? Очевидное решение — перекрасить один из узлов-нарушителей в черный цвет. Попробуем изменить потомка, то есть узел 6. Установите на нем красную стрелку и нажмите кнопку `R/B`. Узел становится черным.

Проблема с красными родителем и потомком решена. К сожалению, появляется другое сообщение об ошибке: черные высоты не одинаковы. Путь от корня до узла 6

содержит три черных узла, а в пути от корня до узла 75 их только два. Правило 4 нарушено. Что же делать?

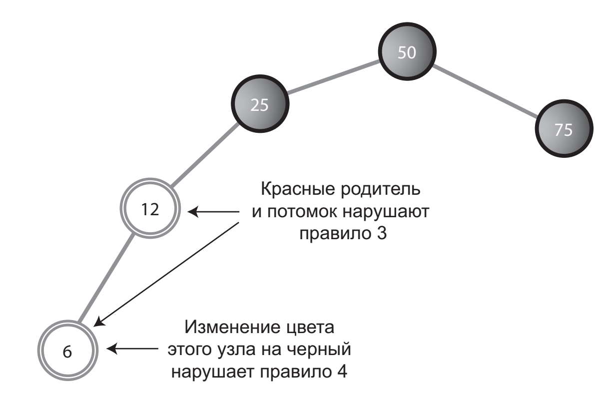


Рис. 9.7. И родитель, и потомок являются красными узлами

Проблема может быть решена с помощью поворота и ряда изменений цветов. О том, как это делается, будет рассказано далее.

Эксперименты продолжаются

Поэкспериментируйте с приложением RBTree Workshop самостоятельно. Вставьте новые узлы и посмотрите, что произойдет. Удастся ли вам добиться сбалансированности дерева, используя повороты и изменения цветов? Гарантирует ли правильность красно-черного дерева его сбалансированность (почти полную)?

Попробуйте вставить серию ключей, упорядоченных по возрастанию (50, 60, 70, 80, 90), затем снова нажмите кнопку `Start` и введите серию, упорядоченную по убыванию (50, 40, 30, 20, 10). Не обращайте внимания на сообщения; вскоре вы поймете, о чем в них говорится. В таких ситуациях в обычном дереве двоичного поиска начинаются неприятности. Удастся ли вам сбалансировать полученное дерево?

Красно-черные правила и сбалансированные деревья

Попробуйте создать дерево, несбалансированное на двух и более уровнях, но при этом являющееся правильным красно-черным деревом. Оказывается, это невозможно. Именно по этой причине красно-черные правила обеспечивают сбалансированность дерева. Если один путь более чем на один узел длиннее другого, он неизбежно либо содержит больше черных узлов (нарушение правила 4), либо содержит два смежных красных узла (нарушение правила 3). Поэкспериментируйте с приложением и убедитесь в том, что это действительно так.

Пустые потомки

Вспомните правило 4: все пути, идущие от корня к любому листовому узлу или пустому потомку, должны содержать одинаковое количество черных узлов. Пустым потомком называется место возможного присоединения потомка к не-листовому узлу (то есть отсутствующий левый потомок узла, имеющий только правого потомка, или наоборот). Таким образом, на рис. 9.8 путь от узла 50 через узел 25 к правому потомку узла 25 (его пустому потомку) содержит только один черный узел, тогда как путь к 6 и 75 содержит два черных узла. Такая конфигурация нарушает правило 4, согласно которому оба пути к листовым узлам должны содержать одинаковое количество черных узлов.

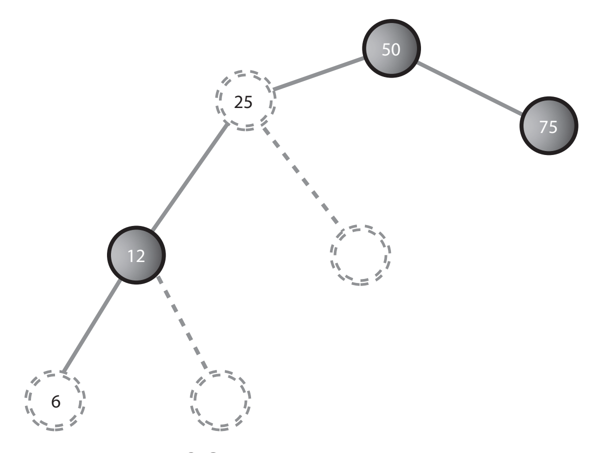


Рис. 9.8. Путь к пустому потомку

Термином «*черная высота*» обозначается количество черных узлов от корня до заданного узла. Так, на рис. 9.8 черная высота узла 50 равна 1, для узла 25 — тоже 1, для узла 12 она равна 2 и т. д.

Повороты

Чтобы сбалансировать дерево, необходимо выполнить физическую перекомпоновку его узлов. Например, если все узлы находятся в левой части дерева, некоторые из них необходимо переместить на правую сторону. Задача решается при помощи поворотов. В этом разделе вы узнаете, что такое повороты и как они выполняются. Поворот должен:

- поднять одни узлы и опустить другие, чтобы обеспечить сбалансированность дерева;
- обеспечить соблюдение характеристик дерева двоичного поиска.

Вспомните, что в дереве двоичного поиска ключи левых потомков любого узла меньше ключа самого узла, а ключи правых потомков больше либо равны ключу

узла. Если бы поворот не поддерживал действительность дерева двоичного поиска, то пользы от него было бы немного; ведь работа алгоритма поиска (см. предыдущую главу) зависит от конфигурации дерева поиска.

Обратите внимание: цветовые правила и изменения в цветах узлов только помогают решить, когда должен выполняться поворот. Возня со сменой цветов мало что дает сама по себе; основная работа выполняется поворотами. Цветовые правила можно сравнить с общими правилами строительства зданий (например, «внешние двери должны открываться вовнутрь»), а повороты — с самими строительными операциями.

Простые повороты

В эксперименте 2 мы опробовали левые и правые повороты. Их достаточно легко себе представить, потому что в них задействованы всего три узла. Давайте проясним некоторые аспекты этого процесса.

Что поворачивается?

Термин «поворот» можно понять неверно. Сами узлы сохраняют исходную ориентацию; изменяются лишь связи между ними. Один узел выбирается в качестве «верхнего» узла поворота. При выполнении правого поворота «верхний» узел смещается вниз и вправо, в позицию своего правого потомка. Его левый потомок смещается вверх, занимая его место. Помните, что верхний узел не является «центром» поворота.

В эксперименте 2 верхним узлом был корень дерева, но, конечно, верхним узлом может быть любой узел, имеющий соответствующего потомка.

Помните о потомках

Не забудьте, что для выполнения правого поворота у верхнего узла должен существовать левый потомок. В противном случае просто не окажется узла, который должен занять верхнюю позицию. Аналогичным образом, при выполнении левого поворота у верхнего узла должен быть правый потомок.

Переходящий узел

Повороты бывают и более сложными, чем в рассмотренном ранее примере с тремя узлами. Нажмите кнопку `Start`, а затем вставьте в дерево (уже содержащее корневой узел 50) узлы со следующими значениями ключей в указанном порядке: 25, 75, 12, 37.

При попытке вставить ключ 12 выводится сообщение о необходимости переключения цветов. Нажмите кнопку `Flip`; цвета родителя и потомка меняются на противоположные. Снова нажмите `Ins`, чтобы завершить вставку узла 12. Наконец, вставьте узел 37. Итоговое дерево выглядит так, как показано на рис. 9.9, *а*.

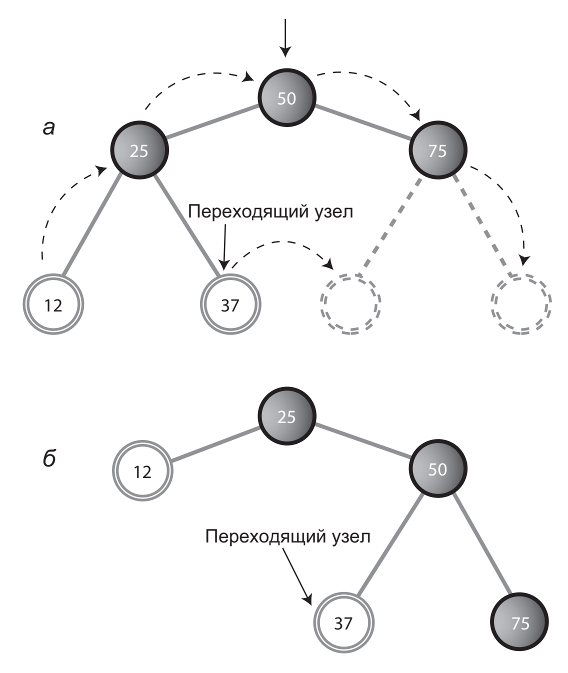


Рис. 9.9. Поворот с переходящим узлом

Теперь попробуем выполнить поворот. Установите стрелку на корневом узле (не забудьте!) и нажмите кнопку `RoR`. Все узлы дерева перемещаются в новые позиции. Узел 12 переходит на место 25 вверх, а узел 50 переходит на место 75 вниз.

Но что это? Узел 37 отсоединился от узла 25, правым потомком которого он был, и стал левым потомком 50. Одни узлы переместились вверх, другие вниз, а узел 37 переместился по горизонтали. Результат показан на рис. 9.9, *б*. Поворот привел к нарушению правила 4; вскоре вы узнаете, как решается эта проблема.

В исходной конфигурации на рис. 9.9, *а* узел 37 называется *внутренним внуком* верхнего узла 50 (соответственно узел 12 — *внешний внук*). Внутренний внук, который является потомком узла, перемещающегося вверх (левый потомок верхнего узла при правом повороте), всегда отсоединяется от родителя и связывается со своим бывшим «дедом».

Перемещения поддеревьев

Мы уже видели, как при поворотах меняется позиция отдельных узлов, однако перемещаться могут целые поддеревья. Нажмите `Start` для создания дерева с корнем 50, а затем вставьте следующую последовательность узлов: 25, 75, 12, 37, 62, 87, 6, 18, 31, 43 (именно в таком порядке). Каждый раз, когда приложение сообщает о необходимости переключения цветов, нажимайте кнопку `Flip`. Полученная конфигурация изображена на рис. 9.10, *а*.

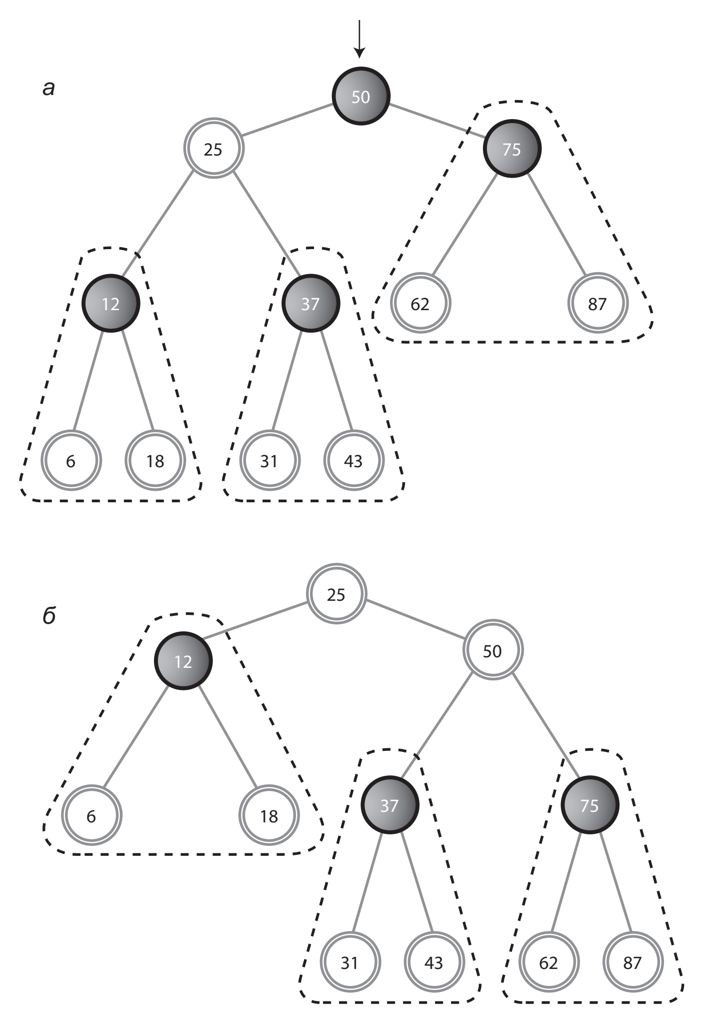


Рис. 9.10. Перемещение поддерева при повороте

Установите стрелку на корневом узле 50, нажмите кнопку RoR. Ого! Многие узлы переместились в новые позиции. Результат показан на рис. 9.10, *б*. Произошло следующее:

- Корневой узел (50) перешел на место правого потомка.
- Левый потомок верхнего узла (25) перешел на место корневого узла.
- Все поддерево с корнем 12 сдвинулось вверх.
- Все поддерево с корнем 37 переместилось на место левого потомка 50.
- Все поддерево с корнем 75 сдвинулось вниз.

В приложении выводится сообщение об ошибке (корень должен быть черным узлом), но на него пока можно не обращать внимания. Чтобы переходить от одной конфигурации к другой, поочередно нажимайте кнопки RoR и RoL (при этом стрелка

должна находиться у верхнего узла). Проследите за тем, что происходит с поддеревьями (особенно с поддеревом, корнем которого является узел 37).

На рис. 9.10 поддеревья выделены пунктирными треугольниками. Обратите внимание: связи между узлами в пределах каждого поддерева не изменяются при поворотах. Все поддерево перемещается как единое целое. Поддеревья при этом не ограничиваются тремя узлами, как в нашем примере. Сколько бы узлов ни содержало поддерево, при повороте все они перемещаются как единое целое.

Люди и компьютеры

К этому моменту вы знаете практически все, что необходимо знать о поворотах. Чтобы выполнить поворот в приложении Workshop, установите стрелку на верхнем узле и нажмите кнопку RoR или RoL. Конечно, в настоящем алгоритме вставки в красно-черном дереве повороты выполняются на программном уровне, без человеческого вмешательства.

Вероятно, человек сможет сбалансировать любое дерево, просто взглянув на него и выполнив соответствующие повороты. Если узел имеет слишком много левых потомков и мало правых, он поворачивается направо, и наоборот.

К сожалению, компьютер не может «просто взглянуть» на структуру дерева — он может только проверить, соответствует ли дерево некоторым простым правилам. Именно такая система формализованных признаков воплощена в форме цветовых обозначений и четырех красно-черных правил.

Вставка узла

Теперь вы знаете все необходимое для того, чтобы понять, как алгоритм вставки красно-черного дерева использует повороты и красно-черные правила для сохранения сбалансированности дерева.

Общая схема процесса вставки

Прежде чем переходить к подробному описанию, мы сначала в общих чертах рассмотрим процесс вставки. Не огорчайтесь, если какие-то подробности останутся неясными; вскоре процесс вставки будет рассмотрен более подробно.

В следующем описании символы X, P и G используются для обозначения связей между узлами. X — узел, нарушающий красно-черные правила. (Иногда символом X обозначается вновь вставленный узел, а иногда потомок при возникновении конфликта типа «красный-красный» между родителем и потомком.)

- X — некоторый узел дерева.
- P — родитель X.
- G — предок X второго уровня (родитель P).

Перемещаясь вниз по дереву в ходе поиска точки вставки, при каждом обнаружении черного узла с двумя красными потомками (нарушение правила 2) выполняется переключение цветов. Иногда переключение создает конфликт типа «красный-красный» (нарушение правила 3). Обозначим красного потомка X, а красного родителя — P. Конфликт разрешается одиночным или двойным поворотом в зависимости от того, является ли X внешним или внутренним внуком G. После переключений и поворотов алгоритм продолжает движение к точке вставки и в конечном итоге вставляет новый узел.

После вставки нового узла X, если узел P черный, вы просто присоединяете новый красный узел. Если узел P красный, возможны два варианта: X является либо внешним, либо внутренним внуком G. Алгоритм выполняет два изменения цветов (подробности чуть позже). Если X является внешним внуком, выполняется один поворот, а если внутренним — два. В результате дерево восстанавливается в сбалансированном состоянии.

Теперь рассмотрим эту краткую схему более подробно. Обсуждение разделено на три части по сложности:

1. Переключения цветов при перемещении вниз.
2. Повороты после вставки узла.
3. Повороты при перемещении вниз.

Если бы три части обсуждались в строго хронологическом порядке, то часть 3 должна была бы рассматриваться до части 2. Однако повороты в нижней части дерева описывать проще, чем повороты в середине, а операции 1 и 2 встречаются чаще операции 3, поэтому мы рассмотрим 2 раньше 3.

Переключения цветов при перемещении вниз

Алгоритм вставки в красно-черное дерево начинает свою работу практически с того же, что и алгоритм вставки в обычное дерево двоичного поиска: он проходит от корня к месту вставки узла, поворачивая налево или направо в зависимости от результата сравнения ключа текущего узла с искомым ключом.

Однако в красно-черном дереве переход к точке вставки усложняется переключениями цветов и поворотами. Переключения были представлены в эксперименте 3; пора рассмотреть их более подробно.

Представьте, что алгоритм вставки перемещается вниз по дереву, поворачивая налево или направо на каждом узле, подыскивая место для вставки нового узла. Чтобы цветовые правила не нарушались, алгоритму приходится выполнять переключение цветов при необходимости. Правило выглядит так: каждый раз, когда алгоритм вставки встречает черный узел с двумя красными потомками, он должен перекрасить потомков в черный цвет, а родителя в красный (если только родитель не является корневым узлом, который всегда остается черным).

Как переключение цветов влияет на красно-черные правила? Для удобства обозначим узел в вершине треугольника (тот, который был черным до переключения) буквой P (от слова *Parent*). Левый и правый потомки P будут обозначаться X1 и X2. Структура поддерева изображена на рис. 9.11, *а*.

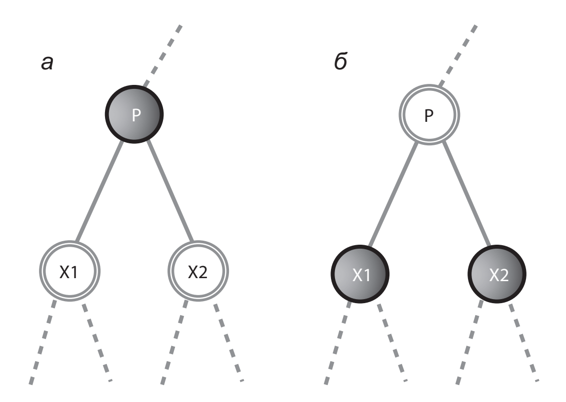


Рис. 9.11. Переключение цветов

Неизменность черных высот

На рис. 9.11, *б* изображены узлы после переключения цветов. Переключение оставляет неизменным количество черных узлов на пути от корня через P к листовым или пустым узлам. Все такие пути проходят через P, а затем либо через X1, либо через X2. До переключения черным является только узел P, поэтому треугольник (состоящий из узлов P, X1 и X2) добавляет в каждый из этих путей один черный узел.

После переключения узел P перестает быть черным, но черными становятся узлы X1 и X2; таким образом, треугольник снова вносит один черный узел в каждый путь, проходящий через него. Таким образом, переключение цветов не может привести к нарушению правила 4.

Переключения цветов полезны, потому что они превращают красные листовые узлы в черные листовые узлы. Это упрощает добавление новых красных узлов без нарушения правила 3.

Нарушение правила 3

Хотя переключение цветов не нарушает правила 4, правило 3 (узел и его родитель не могут одновременно быть красными) может его нарушить. Если родитель P является черным узлом, превращение P из черного узла в красный не создает проблем. Но если родитель P является красным узлом, после изменения цвета появятся два красных узла подряд.

Эту проблему необходимо решить до того, как двигаться дальше вниз для вставки нового узла. Как вы вскоре увидите, она решается посредством поворота.

Корневой узел

Как насчет корня дерева? Напомню, что при переключении цветов корня и двух его потомков сам корень остается черным, чтобы предотвратить нарушение правила 2. Влияет ли это на другие красно-черные правила? Естественно, переключение не породит конфликтов типа «красный-красный», потому что ни один узел не был окрашен в красный цвет. Таким образом, правило 3 не нарушается. Кроме того,

поскольку в каждый путь заведомо входит корневой узел и один из его двух потомков, черная высота каждого пути увеличивается на одну и ту же величину, а именно на единицу. Таким образом, правило 4 тоже не нарушается.

Вставка

Когда вы доберетесь до соответствующей позиции дерева, при необходимости выполняя переключения цветов (и повороты), можно выполнить вставку нового узла — это делается точно так же, как в обычном дереве двоичного поиска (см. предыдущую главу).

Тем не менее работа еще не закончена.

Повороты после вставки узла

Вставка нового узла может привести к нарушению красно-черных правил. Таким образом, после вставки необходимо проверить выполнение правил и при необходимости принять соответствующие меры.

Вспомните, что только что вставленный узел (который мы будем называть X) всегда окрашен в красный цвет. Как показано на рис. 9.12, узел X может находиться в разных позициях относительно P и G.

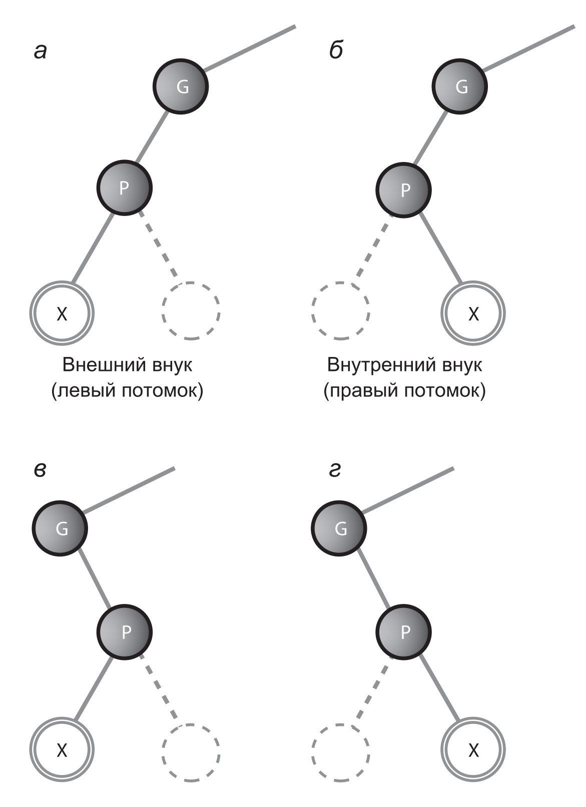


Рис. 9.12. Варианты вставки узла

Напомним, что узел X называется внешним внуком, если он находится на одной стороне с родителем P по отношению к родителю последнего G. Таким образом, X является внешним внуком, если он либо является левым потомком P, а P является левым потомком G (рис. 9.12, *а*), либо он является правым потомком P, а P является правым потомком G (рис. 9.12, *г*). И наоборот, X называется внутренним внуком, если он находится на противоположной стороне с родителем P по отношению к родителю последнего G (рис. 9.12, *б* и *в*).

Обилие вариантов на рис. 9.12 (далее они будут называться *дисимметричными* вариантами) — одна из причин, усложняющих реализацию красно-черных алгоритмов вставки.

Действия, выполняемые для восстановления красно-черных правил, определяются цветами и конфигурацией узла X и его «родственников». Как ни странно, существует всего три основных варианта конфигурации (не считая уже упомянутых вариантов с внутренними/внешними внуками).

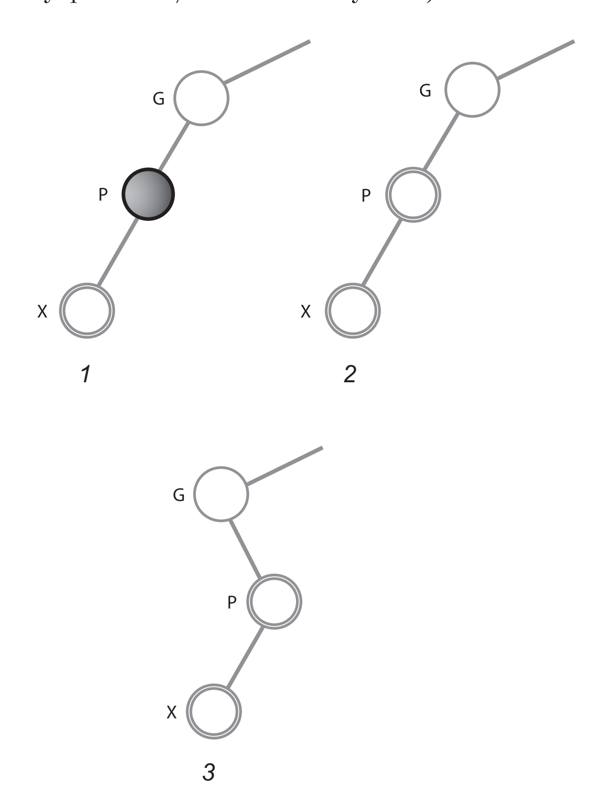


Рис. 9.13. Варианты конфигурации узлов

В каждом из возможных вариантов выбирается свой способ сохранения красно-черной правильности, приводящий к созданию сбалансированного дерева. Сначала мы кратко рассмотрим все три варианта конфигурации узлов, а затем обсудим каждый из них более подробно в соответствующем разделе. Варианты изображены на рис. 9.13 (помните, что X всегда является красным узлом):

1. P — черный узел.
2. P — красный узел, а X — внешний внук G.
3. P — красный узел, а X — внутренний внук G.

Может показаться, что этот список не исчерпывает всех возможных вариантов. Мы еще вернемся к этому вопросу позднее, после анализа этих трех случаев.

Вариант 1. P — черный узел

Если узел P черный, все просто. Только что вставленный узел всегда красный. Если его родитель черный, то конфликта «красный-красный» (правило 3) нет, а количество черных узлов не увеличивается (правило 4). Следовательно, красно-черные правила не нарушаются. Больше делать ничего не нужно, вставка завершена.

Вариант 2. P — красный узел, а X — внешний

Если узел P красный, а X является внешним внуком, потребуется один поворот и несколько изменений цветов. Давайте смоделируем ситуацию в приложении RBTree Workshop и посмотрим, о чем идет речь. Начните со стандартного корня 50 и вставьте узлы 25, 75 и 12. Перед вставкой 12 необходимо выполнить переключение цветов.

Теперь вставьте узел 6 — это наш новый узел X. Полученное дерево изображено на рис. 9.14, *а*. В приложении Workshop выводится сообщение об ошибке — и родитель, и потомок являются красными узлами. Необходимо принять меры к исправлению дерева.

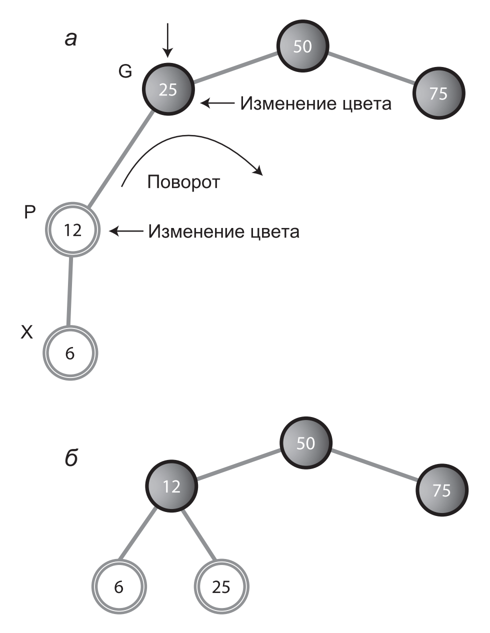


Рис. 9.14. P — красный узел, а X — внешний внук G

В этой ситуации для восстановления красно-черной правильности дерева (а следовательно, его сбалансированности) достаточно выполнить три операции:

1. Переключить цвет предка G (25 в данном примере).
2. Переключить цвет родителя P (12).
3. Выполнить поворот с верхним узлом G (25) в направлении, поднимающем X (6). В нашем случае это правый поворот.

Как вы уже знаете, для изменения цветов нужно установить стрелку на узле и нажать кнопку `R/B`. Чтобы выполнить сдвиг вправо, установите стрелку на узле и нажмите кнопку `RoR`. Когда все три действия будут выполнены, приложение Workshop сообщит, что дерево является правильным красно-черным деревом. Кроме того, дерево становится более сбалансированным, чем прежде (рис. 9.14, *б*).

В этом примере узел X был внешним внуком и левым потомком. Также существует симметричная ситуация, когда X является внешним внуком, но правым потомком. Попробуйте создать дерево 50, 25, 75, 87, 93 (выполните переключение цветов при необходимости). Чтобы исправить дерево, измените цвета 75 и 87 и выполните поворот влево с верхним узлом 75. Дерево снова сбалансировано.

Вариант 3. P — красный узел, а X — внутренний

Если узел P красный, а X является внутренним внуком, потребуется два поворота и несколько изменений цветов. Рассмотрим пример: запустите приложение RBTree Workshop и создайте дерево 50, 25, 75, 12, 18. (Как и в предыдущем случае, перед вставкой 12 необходимо выполнить переключение цветов.) Результат показан на рис. 9.15, *а*.

Обратите внимание: узел 18 является внутренним внуком. И он, и его родитель являются красными узлами, поэтому в приложении снова выводится сообщение об ошибке.

Сбалансировать эту конфигурацию немного сложнее. Если попытаться выполнить поворот вправо с верхним узлом G (25), как в варианте 2, внутренний внук X (18) сместится по горизонтали вместо перемещения вверх, поэтому сбалансированность дерева не улучшится. (Попробуйте выполнить эту операцию с верхним узлом 12, а затем выполните обратный поворот для возврата к исходной конфигурации.) Требуется другое решение.

Если X является внутренним внуком, проблема решается выполнением двух поворотов вместо одного. Первый поворот превращает X из внутреннего внука во внешнего, как показано на рис. 9.15, *а* и *б*. Теперь ситуация стала ближе к варианту 1, и мы можем применить тот же поворот с тем же верхним узлом-предком, как и прежде. Результат показан на рис. 9.15, *в*.

Также необходимо перекрасить узлы. Мы сделаем это перед выполнением каких-либо поворотов (на самом деле порядок может быть любым, но если заняться перекрашиванием после поворотов, нам будет труднее описать, о каких узлах идет речь). Это делается так:

1. Измените цвет предка X (25 в данном примере).
2. Измените цвет X (не его родителя; в нашем примере это узел 18).

3. Выполните поворот, верхним узлом которого является родительский узел P (то есть узел 12), в направлении, поднимающем X (левый поворот в нашем примере).
4. Снова выполните поворот, верхним узлом которого является предок X (25), в направлении, поднимающем X (правый поворот).

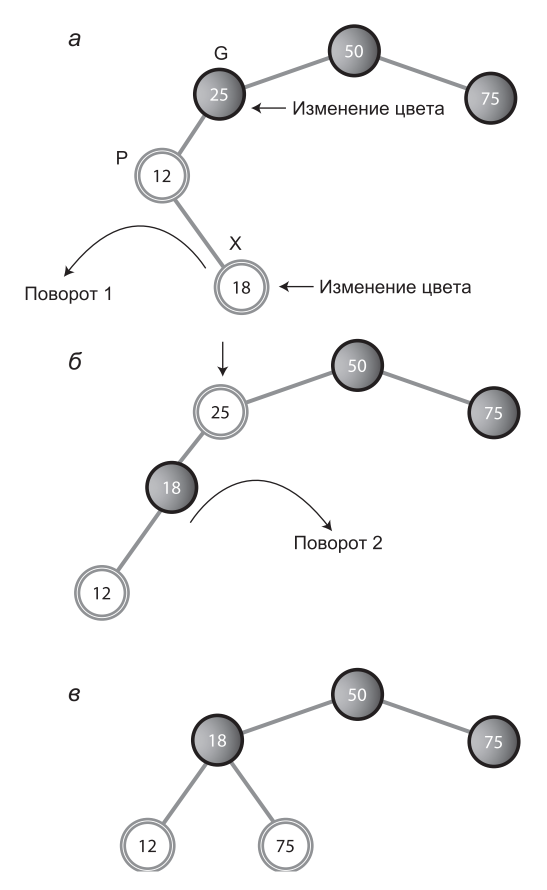


Рис. 9.15. P — красный узел, а X — внутренний внук G

Повороты и изменение цветов восстанавливают красно-черную правильность дерева, а также обеспечивают его сбалансированность (насколько это возможно). Как и в варианте 2, существует аналогичная ситуация, в которой P является не левым, а правым потомком G.

Как насчет других вариантов?

Действительно ли три рассмотренных нами варианта исчерпывают все возможные ситуации? Допустим, что у X имеется «братский» узел S — другой потомок P. Этот сценарий может усложнить повороты, необходимые для вставки X. Для черного

узла P вставка X не создает проблем (вариант 1). Если узел P красный, то оба его потомка должны быть черными (чтобы избежать нарушения правила 3). Он не может иметь единственного черного потомка S, потому что это приведет к разным значениям черных высот S и пустого потомка. Но так как мы знаем, что узел X красный, из этого можно сделать вывод, что у X не может быть «брата», если только P не является красным узлом.

Другая возможность: у узла G, предка P, имеется потомок U, «брат» P и «дядя» X. В этой ситуации необходимые повороты тоже усложняются. Но как мы уже видели, для черного узла P вставка X поворотов не требует; соответственно будем считать, что узел P красный. В этом случае узел U тоже должен быть красным (иначе черная высота пути, идущего от G к P, будет отличаться от черной высоты пути, идущего от G к U). Но к черному родителю с двумя красными потомками на пути вниз применяется переключение цветов, поэтому такая ситуация тоже невозможна.

Итак, три рассмотренных варианта исчерпывают весь спектр возможных случаев (не считая того, что в вариантах 2 и 3 узел X может быть правым или левым потомком).

Что делают переключения цветов

Допустим, в результате поворотов и соответствующих изменений цветов в дереве возникают другие нарушения красно-черных правил. Нетрудно представить себе неприятную ситуацию: вам приходится возвращаться вверх по дереву и устранять нарушения посредством поворотов и переключения цветов.

К счастью, такая ситуация невозможна. Переключения цветов на пути вниз устраняют ситуации, в которых повороты могут привести к каким-либо нарушениям правил на более высоких уровнях дерева. Они гарантируют, что один или два поворота восстановят красно-черную правильность во всем дереве. Доказательство этого утверждения выходит за рамки книги, но оно существует.

Благодаря переключению цветов на пути сверху вниз вставка в красно-черных деревьях отличается большей эффективностью по сравнению с другими сбалансированными деревьями (например, AVL-деревьями). Они гарантируют, что одного прохода сверху вниз по дереву будет достаточно.

Повороты при перемещении вниз

Переходим к последней из трех операций, связанных со вставкой узлов: поворотам на пути вниз к точке вставки. Как уже упоминалось, хотя эта операция рассматривается последней, выполняется она до вставки узла. Ее рассмотрение было отложено только потому, что повороты проще описывать для только что вставленного узла, чем для узлов в середине дерева.

В описании переключений цветов в процессе вставки упоминалось о том, что переключение может привести к нарушению правила 3 (родитель и потомок не могут одновременно быть красными). Также было отмечено, что нарушение исправляется посредством поворота.

При перемещении вниз возможны две ситуации, соответствующие вариантам 2 и 3 в приведенном ранее описании фазы вставки. Узел-нарушитель может быть внешним или внутренним внуком (в ситуации, соответствующей варианту 1, действия не требуются).

Внешний внук

Начнем с примера, в котором узел-нарушитель является внешним внуком Под «узлом-нарушителем» понимается потомок в паре «родитель-потомок», породившей конфликт типа «красный-красный».

Начните новое дерево с корневым узлом 50, вставьте в него следующие узлы: 25, 75, 12, 37, 6 и 18. При вставке узлов 12 и 6 необходимо выполнить преобразования цветов. Нажмите кнопку `Flip`. Переключение выполняется, но теперь в сообщении говорится о конфликте типа «красный-красный» — речь идет об узле 25 и его потомке 12. Полученное дерево изображено на рис. 9.16, *а*.

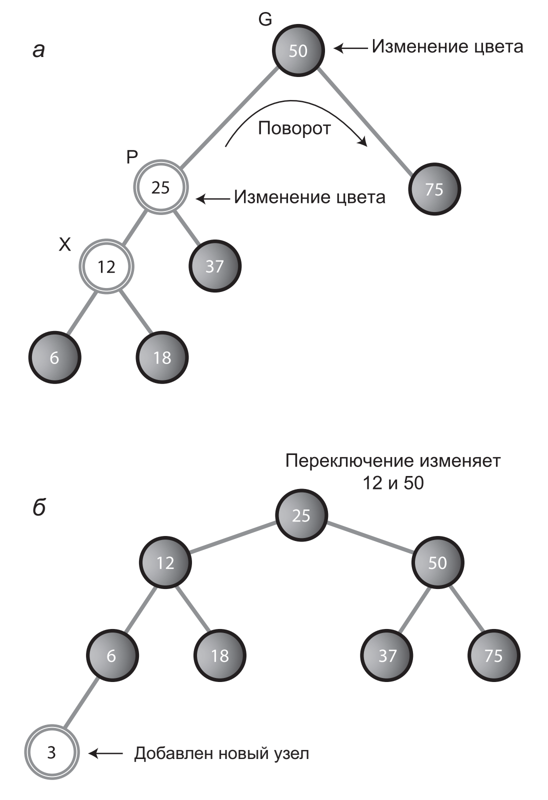


Рис. 9.16. Внешний внук при перемещении вниз

Процедура, используемая для исправления этого нарушения, сходна с аналогичной операцией, выполняемой после вставки для внешнего внука (см. выше), — необходимо выполнить два переключения цветов и один поворот. Чтобы описывать

происходящее в тех же терминах, как при вставке узла, мы будем называть верхний узел переключаемого треугольника (12 в данном случае) X. Возможно, это выглядит немного странно, потому что в предыдущих описаниях узел X вставлялся в дерево, а здесь он даже не является листовым. Тем не менее повороты при перемещении вниз могут выполняться в любой позиции дерева.

Родитель X (25 в нашем примере) будет обозначаться P, а родитель P, то есть предок X (50), будет обозначаться G. Исправление выполняется так же, как в рассмотренном ранее варианте 2:

1. Переключите цвет предка G (50 в данном примере). Не обращайте внимания на сообщение о том, что корень должен быть черным.
2. Переключите цвет родителя P (25).
3. Выполните поворот с верхним узлом G (50) в направлении, поднимающем X (6). В нашем случае это правый поворот.

И дерево неожиданно становится сбалансированным! К тому же оно обретает приятную симметричность. На первый взгляд кажется чудом, но в действительности это лишь результат выполнения цветовых правил. Теперь узел со значением 3 может быть вставлен обычным способом. Поскольку узел 6, с которым он связан, является черным, вставка не вызывает никаких сложностей. Потребуется всего одно переключение цветов (узел 50). На рис. 9.16, *б* показано, как выглядит дерево после вставки узла 3.

Внутренний внук

Если при возникновении конфликта типа «красный-красный» при перемещении вниз узел X является внутренним внуком, для исправления ситуации потребуются два поворота. Выполняемые операции аналогичны операциям для внутреннего внука в фазе после вставки (мы назвали эту ситуацию «вариант 3»).

Нажмите кнопку `Start` в приложении RBTree Workshop, чтобы создать дерево с корнем 50. Вставьте узлы 25, 75, 12, 37, 31 и 43. Перед вставкой узлов 12 и 31 необходимо выполнить переключение (от узла 37). Но при выполнении переключения оба узла 37 и 25 становятся красными, и приложение выводит соответствующее сообщение об ошибке. Пока не нажимайте кнопку `Ins`.

В этой ситуации обозначение G соответствует узлу 50, P — узлу 25, а X — узлу 37 (рис. 9.17, *а*).

Устранение конфликта типа «красный-красный» потребует тех же двух изменений цветов и двух поворотов, что и в варианте 3:

1. Измените цвет предка G (50 в данном примере). Не обращайте внимания на сообщение о том, что корень должен быть черным.
2. Измените цвет X (37).
3. Выполните поворот, верхним узлом которого является родительский узел P (25). Результат показан на рис. 9.17, *б*.
4. Снова выполните поворот, верхним узлом которого является предок X (50), в направлении, поднимающем X (правый поворот).

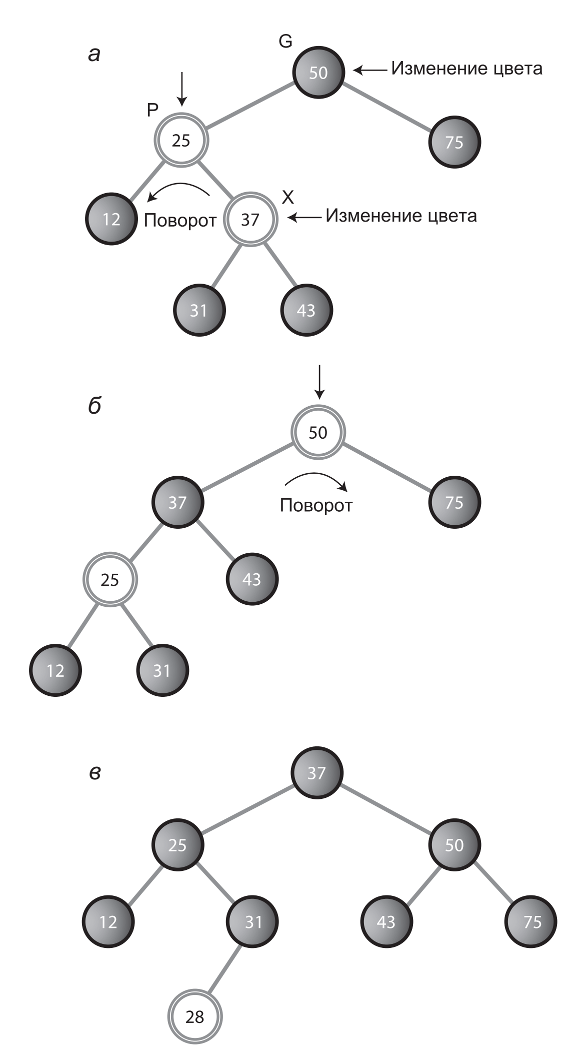


Рис. 9.17. Внутренний внук при перемещении вниз

Теперь можно вставить узел 28. При вставке переключение окрашивает узлы 25 и 50 в черный цвет. Результат показан на рис. 9.17, *в*.

На этом наше обсуждение сохранения красно-черной правильности (а следовательно, сбалансированности) дерева при вставке подходит к концу.

Удаление

Как вы, вероятно, помните, операция для обычного дерева двоичного поиска программируется намного сложнее, чем операция вставки. Это относится и к красно-черным деревьям, но операция удаления дополнительно усложняется необхо-

димостью восстановления красно-черной правильности дерева после удаления узла.

Более того, процесс удаления усложняется настолько, что многие программисты стараются различными способами обойти его. В одном из способов (используемом и для обычных двоичных деревьев) узел помечается как удаленный без его фактического удаления. Любой алгоритм поиска, обнаруживающий этот узел, знает, что сообщать о нем никому не нужно. Такое решение работает во многих ситуациях, особенно если удаления выполняются относительно редко. Как бы то ни было, мы не будем рассматривать процесс удаления. Если вас интересует эта тема, обращайтесь к приложению С.

Эффективность красно-черных деревьев

В красно-черных деревьях, как и в обычных деревьях двоичного поиска, операции поиска, вставки и удаления выполняются за время $O(\log_2 N)$. Время поиска в красно-черных деревьях практически неотличимо от времени поиска в обычном дереве, потому что при поиске красно-черные характеристики не используются. Единственное отличие — небольшое увеличение затрат памяти на хранение признака красно-черного цвета (логическая переменная).

А конкретно, по данным Седжвика (см. приложение Б), на практике поиск в красно-черном дереве в среднем требует около $\log_2 N$ сравнений, причем можно доказать, что поиск не может требовать более $2 \times \log_2 N$ сравнений.

Время удаления и вставки возрастает с постоянным коэффициентом из-за необходимости выполнения переключений и поворотов при перемещении вниз и в точке вставки. В среднем при вставке выполняется около одного поворота. Таким образом, сложность вставки тоже составляет $O(\log_2 N)$, но операция выполняется медленнее вставки в обычном двоичном дереве.

Так как в большинстве приложений количество операций поиска обычно превышает количество вставок и удалений, вероятно, использование красно-черного дерева вместо обычного не приведет к сколько-нибудь заметному снижению эффективности. И конечно, к преимуществам красно-черного дерева следует отнести и то, что работа с отсортированными данными не вырождается до низкой производительности $O(N)$.

Реализация красно-черного дерева

Чтобы реализовать вставку для красно-черных деревьев, нужно сделать совсем немного (да, это была ирония): написать код выполнения операций, описанных в предыдущих разделах. Как упоминалось ранее, демонстрация и описание этого кода выходит за рамки книги. Впрочем, мы в общих чертах упомянем важнейшие моменты реализации.

В класс `Node` необходимо включить поле для хранения цвета (которое может относиться к типу `boolean`).

Код вставки можно позаимствовать из программы `tree.java` (листинг 8.1 главы 8). При перемещении вниз к точке вставки проверьте, является ли текущий узел черным узлом с двумя красными потомками. Если это условие выполнено, измените цвета всех трех узлов (если только родитель не является корневым узлом — в этом случае он остается черным).

После переключения цветов проверьте, что правило 3 не нарушено. Если оно нарушается, выполните соответствующие повороты: один для внешнего внука, два для внутреннего.

Достигнув листового узла, вставьте новый узел, как в программе `tree.java`, убедившись в том, что этот узел красный. Снова проверьте конфликты типа «красный-красный» и выполните все необходимые повороты.

Как ни странно, программной реализации не нужно вычислять черные высоты разных частей дерева (хотя, возможно, эта информация пригодится в ходе отладки). Проверять нужно только нарушения правила 3, красного родителя с красным потомком, которые могут осуществляться локально (в отличие от черных высот из правила 4, требующих более сложных подсчетов).

Если выполнить переключения, изменения и повороты цветов, о которых говорилось ранее, черные высоты узлов выравниваются автоматически, а дерево остается сбалансированным. Приложение RBTree Workshop сообщает об ошибках черных высот только потому, что оно не заставляет пользователя правильно выполнять алгоритм вставки.

Другие сбалансированные деревья

Из всех разновидностей сбалансированных деревьев первым появилось AVL-дерево, названное по фамилиям своих изобретателей: Адельсона-Вельского и Ландиса. В AVL-деревьях в каждом узле хранится дополнительная информация: разность весов левого и правого поддеревьев. Эта разность не может быть больше единицы. Иначе говоря, высота левого поддерева узла не может более чем на единицу отличаться от высоты правого поддерева.

После вставки проверяется корень самого нижнего поддерева, в котором был вставлен новый узел. Если высоты потомков отличаются более чем на 1, выполняется одиночный или двойной поворот для выравнивания высот. Затем алгоритм двигается вверх и проверяет узел, находящийся над ним, с выравниванием высот в случае необходимости. Далее проверка продолжается на всем пути вверх до корня.

Поиск в AVL-дереве выполняется за время $O(\log N)$, так как дерево заведомо сбалансировано. Однако вставка (и удаление) узла требует двух проходов по дереву: до точки вставки для восстановления баланса дерева, поэтому AVL-деревья по эффективности уступают красно-черным деревьям и на практике используются не так часто.

Другую важную категорию сбалансированных деревьев составляют *многопутевые* (multiway) *деревья*, в которых каждый узел может иметь более двух потомков. Одна из разновидностей многопутевых деревьев — деревья 2-3-4 — рассматрива-

ется в следующей главе. Один из недостатков многопутевых деревьев заключается в возрастании затрат памяти, поскольку в каждом узле должны храниться ссылки на каждого из его потомков.

Итоги

- Дерево двоичного поиска должно быть сбалансированным, поскольку это гарантирует минимальное время поиска заданного узла.
- При вставке заранее отсортированных данных создается дерево с максимальной разбалансированностью, время поиска в котором составляет $O(N)$.
- В красно-черной схеме балансировки каждому узлу присваивается новая характеристика: цвет (красный или черный).
- Допустимое взаиморасположение узлов разных цветов определяется набором правил, называемых красно-черными правилами.
- Эти правила применяются при вставке (и удалении) узлов.
- Переключение цветов превращает черный узел с двумя красными потомками в красный узел с двумя черными потомками.
- При выполнении поворота один узел считается верхним.
- При правом повороте верхний узел перемещается в позицию правого потомка, а левый потомок занимает его прежнее место.
- При левом повороте верхний узел перемещается в позицию левого потомка, а правый потомок занимает его прежнее место.
- В ходе определения позиции для вставки нового узла выполняются переключения цветов, а в некоторых случаях и повороты. Переключение просто возвращает дерево к состоянию красно-черной правильности после выполнения вставки.
- После вставки нового узла снова производится проверка возможных конфликтов типа «красный-красный». Если будет обнаружено нарушение, выполняются повторы, возвращающие дерево к состоянию красно-черной правильности.
- В результате этих исправлений дерево становится сбалансированным (или по крайней мере ограничивается минимальной несбалансированностью).
- Добавление красно-черной балансировки в двоичное дерево лишь незначительно ухудшает среднее быстродействие и избегает худшего сценария со вставкой уже изначально отсортированных данных.

Вопросы

Следующие вопросы помогут читателю проверить качество усвоения материала. Ответы на них приведены в приложении В.

1. Сбалансированность деревьев двоичного поиска предотвращает снижение производительности при вставке __________ данных.

2. В сбалансированном дереве:
   a) может потребоваться изменение структуры дерева при поиске;
   b) пути от корня до всех листовых узлов имеют примерно одинаковую длину;
   c) все левые поддеревья имеют такую же высоту, как и все правые поддеревья;
   d) высота всех поддеревьев жестко контролируется.
3. Красно-черные правила требуют возможной перестановки узлов дерева для обеспечения его сбалансированности (Да/Нет).
4. Пустым потомком называется:
   a) потомок, который не существует, но будет создан на следующем шаге;
   b) потомок, не имеющий собственных потомков;
   c) один из двух потенциальных потомков листового узла, в котором будет выполнена операция вставки;
   d) несуществующий левый потомок узла, у которого имеется правый потомок (или наоборот).
5. Какое из следующих утверждений *не является* красно-черным правилом?
   a) все пути от корня к узлу или пустому потомку должны содержать одинаковое количество черных узлов;
   b) если узел черный, то его потомки должны быть красными;
   c) корень всегда окрашен в черный цвет;
   d) все три предшествующих утверждения являются красно-черными правилами.
6. Для восстановления сбалансированности дерева применяются две операции: ________ и ________ .
7. Только что вставленные узлы всегда окрашены в ________ цвет.
8. Какое из следующих действий *не выполняется* при повороте?
   a) Перестановка узлов для восстановления характеристик дерева двоичного поиска.
   b) Изменение цвета узлов.
   c) Проверка выполнения красно-черных правил.
   d) Попытка восстановления сбалансированности дерева.
9. «Переходящий» узел или поддерево изначально является _________, а при перемещении становится ________ , и наоборот.
10. Какое из следующих утверждений *ложно*? Необходимость в выполнении поворотов может возникнуть:
    a) перед вставкой узла;
    b) после вставки узла;
    c) во время поиска позиции вставки;
    d) при поиске узла с заданным ключом.

11. При переключении изменяются цвета _______ и _______ .
12. Внешний внук:
    a) находится на другой стороне от своего родителя, чем его родитель по отношению к своему «брату»;
    b) находится на той же стороне от своего родителя, что и его родитель по отношению к своему родителю;
    c) является левым потомком правого потомка (или наоборот);
    d) находится на противоположной стороне от своего родителя, чем его «брат» по отношению к их «предкам».
13. Если один поворот выполняется сразу же после другого, они выполняются в обратных направлениях (Да/Нет).
14. Два поворота необходимы в ситуации, когда:
    a) узел является внутренним внуком, а родитель окрашен в красный цвет;
    b) узел является внуком, а родитель окрашен в черный цвет;
    c) узел является внешним внуком, а родитель окрашен в красный цвет;
    d) узел является внешним внуком, а родитель окрашен в черный цвет.
15. Удаление в красно-черном дереве может потребовать некоторого изменения структуры дерева (Да/Нет).

Упражнения

Упражнения помогут вам глубже усвоить материал этой главы. Программирования они не требуют.

1. Постройте диаграмму активности (или блок-схему) всех вариантов взаимного расположения и цветов узлов, возможных при вставке нового узла в красно-черном дереве (а также действий, выполняемых в каждой ситуации для вставки узла).
2. Воспроизведите в приложении RBTree Workshop все ситуации, описанные в эксперименте 1, и выполните инструкции по вставке.
3. Выполните достаточное количество вставок и убедитесь, что при точном соблюдении красно-черных правил 1, 2 и 3 правило 4 выполняется автоматически.

ПРИМЕЧАНИЕ

Так как в этой главе примеры кода отсутствуют, поэтому включать в нее какие-либо программные проекты было бы нелогично. Если хотите взяться за действительно нетривиальную задачу — реализуйте красно-черное дерево. В качестве отправной точки можно взять программу tree.java (см. листинг 8.1) из главы 8.

Глава 10

Деревья 2-3-4

В двоичном дереве каждый узел содержит один элемент данных и может иметь до двух потомков. Дерево с большим количеством элементов данных и потомков называется *многопутевым* деревом. *Деревья 2-3-4*, которым посвящена первая часть этой главы, представляют собой многопутевые деревья, у которых каждый узел может иметь до четырех потомков и трех элементов данных.

Деревья 2-3-4 представляют интерес по нескольким причинам. Во-первых, это сбалансированные деревья, которые ведут себя как красно-черные деревья. Они обладают чуть меньшей эффективностью, чем красно-черные деревья, но проще программируются. Во-вторых (что самое важное), они упрощают изучение B-деревьев.

B-дерево представляет собой другую разновидность многопутевых деревьев, особенно удобную для организации данных во внешней памяти (то есть за пределами оперативной памяти; обычно под внешней памятью понимается дисковое устройство). Узел B-дерева может иметь десятки и даже сотни потомков. Внешнее хранение данных и B-деревья рассматриваются в конце главы.

Знакомство с деревьями 2-3-4

В этом разделе рассматриваются характеристики деревьев 2-3-4. Позднее вы увидите, как деревья 2-3-4 моделируются в приложениях Workshop и как запрограммировать дерево 2-3-4 на языке Java. Также будет рассмотрена неожиданно тесная связь деревьев 2-3-4 с красно-черными деревьями.

На рис. 10.1 изображено небольшое дерево 2-3-4. Каждый узел (в форме прямоугольника с закругленными краями) может содержать один, два или три элемента данных.

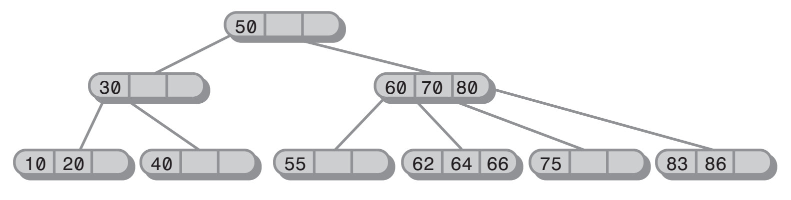


Рис. 10.1. Дерево 2-3-4

У трех верхних узлов имеются потомки, а все шесть узлов из нижнего ряда являются листовыми, то есть по определению не имеют потомков. В дереве 2-3-4 все листовые узлы всегда находятся на одном уровне.

Почему деревья 2-3-4 так называются?

Цифры 2, 3 и 4 в названии дерева обозначают количество связей с потомками, которые могут содержаться в заданном узле. Для не-листовых узлов возможны три конфигурации:

- Узел с одним элементом данных всегда имеет двух потомков.
- Узел с двумя элементами данных всегда имеет трех потомков.
- Узел с тремя элементами данных всегда имеет четырех потомков.

Короче говоря, количество потомков у не-листового узла всегда на единицу больше количества элементов данных. Или если выразить это соотношение в виде формулы, соотношение количества потомков L с количеством элементов данных D выглядит так:

$$L = D + 1.$$

Это важнейшее соотношение определяет структуру деревьев 2-3-4. С другой стороны, листовые деревья не имеют потомков, но могут содержать один, два или три элемента данных. Пустые узлы запрещены.

Поскольку узлы дерева 2-3-4 могут содержать до четырех потомков, такое дерево называется *многопутевым узлом порядка 4*.

Почему дерево 2-3-4 не называется деревом 1-2-3-4? Разве узел не может иметь только одного потомка, как узлы двоичных деревьев? Двоичное дерево (см. главу 8, «Двоичные деревья» и главу 9, «Красно-черные деревья») можно рассматривать как многопутевое дерево порядка 2, потому что каждый узел может иметь до двух потомков. Однако между двоичными деревьями и деревьями 2-3-4 существует важное различие (помимо максимального количества узлов): в двоичном дереве узел может иметь до двух потомков. Одна связь с левым или правым потомком абсолютно допустима; другая ссылка при этом равна `null`.

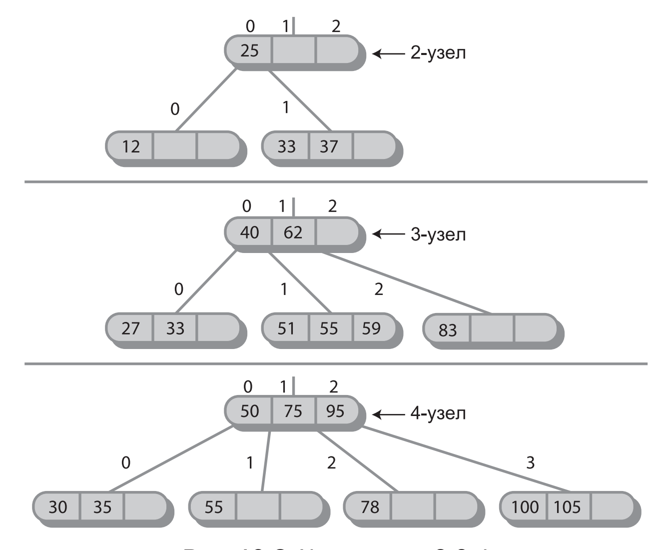


Рис. 10.2. Узлы дерева 2-3-4

В деревьях 2-3-4 узлы с одной связью невозможны. Узел с одним элементом данных всегда должен иметь две связи (если только узел не является листовым — в этом случае он вообще не имеет связей).

Возможные варианты представлены на рис. 10.2. Узел с двумя связями называется 2-узлом, узел с тремя связями называется 3-узлом, а узел с четырьмя связями называется 4-узлом, но такого понятия, как 1-узел, не существует.

Структура дерева 2-3-4

Для удобства мы пронумеруем элементы данных в узле от 0 до 2, а ссылки на потомков — от 0 до 3, как показано на рис. 10.2. Элементы данных узла упорядочиваются по возрастанию ключа; обычно значения располагаются слева направо.

Важным аспектом любой древовидной структуры является связь ее ссылок с ключами элементов данных. В двоичном дереве все потомки с ключом, меньшим ключа узла, содержатся в поддереве, корнем которого является левый потомок узла; соответственно все потомки с ключом, большим ключа узла, содержатся в поддереве, корнем которого является правый потомок. В дереве 2-3-4 действует тот же принцип, но в немного расширенном виде:

- У всех потомков поддерева, корнем которого является узел 0, ключи меньше ключа 0.
- У всех потомков поддерева, корнем которого является узел 1, ключи больше ключа 0, но меньше ключа 1.
- У всех потомков поддерева, корнем которого является узел 2, ключи больше ключа 1, но меньше ключа 2.
- У всех потомков поддерева, корнем которого является узел 3, ключи больше ключа 2.

Отношения наглядно представлены на рис. 10.3. Дубликаты в деревьях 2-3-4 обычно запрещены, поэтому беспокоиться о равенстве ключей при сравнении не нужно.

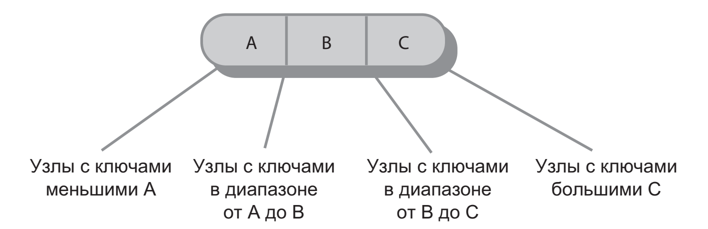


Рис. 10.3. Ключи и потомки

Вернемся к дереву на рис. 10.1. Как и во всех деревьях 2-3-4, листья в нем располагаются на одном уровне (нижний ряд). Узлы верхних уровней часто бывают неполными, то есть содержат только один или два элемента данных вместо трех.

Также заметьте, что дерево сбалансировано. Баланс сохраняется даже при вставке данных, упорядоченных по возрастанию (или убыванию). Как вы вскоре увидите, самобалансируемость дерева 2-3-4 обусловлена новым способом вставки данных.

Поиск в дереве 2-3-4

Поиск элемента данных с заданным ключом напоминает поиск в двоичном дереве. Алгоритм начинает с корня дерева и вплоть до обнаружения ключа выбирает ссылку, ведущую к поддереву с соответствующим диапазоном значений.

Например, чтобы провести поиск элемента данных с ключом 64 в дереве на рис. 10.1, следует начать с корня. При сравнении с корневым ключом элемент не находится. Так как 64 больше 50, алгоритм переходит к потомку 1, которого мы обозначим 60/70/80. (Стоит напомнить, что потомок 1 является правым, потому что нумерация потомков и связей начинается слева.) В этом узле искомый элемент также отсутствует, поэтому необходимо перейти к следующему потомку. На этот раз 64 больше 60, но меньше 70, поэтому алгоритм снова переходит к потомку 1. На этот раз искомый элемент находится в узле 62/64/66.

Вставка

Новые элементы данных всегда вставляются в листьях, находящихся в нижнем ряду дерева. Если узлы вставляются в узлах с потомками, то количество потомков необходимо изменить для сохранения структуры дерева — согласно правилу, гласящему, что количество потомков должно быть на единицу больше количества элементов данных в узле.

В одних случаях вставка в дереве 2-3-4 выполняется легко, в других она основательно усложняется. Процесс всегда начинается с поиска соответствующего листового узла. Если в процессе поиска полные узлы не обнаружены, то вставка выполняется легко — при достижении подходящего листового узла новый элемент данных просто вставляется в него. На рис. 10.4 элемент данных с ключом 18 вставляется в дерево 2-3-4.

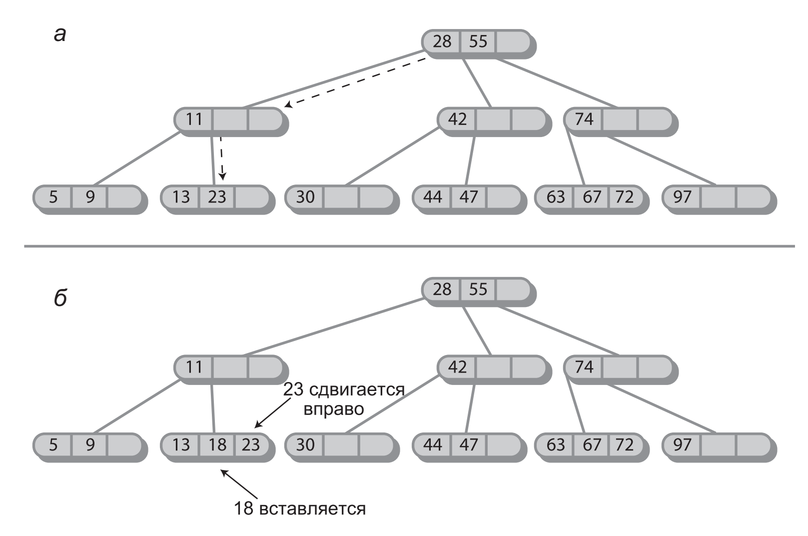


Рис. 10.4. Вставка без разбиений: *а* — до вставки; *б* — после вставки

Вставка может потребовать перемещения одного или двух других элементов данных в узлах, чтобы после вставки нового элемента ключи следовали в правильном порядке. В приведенном примере элемент 23 сдвигается вправо, чтобы освободить место для 18.

Разбиение узлов

Если на пути вниз к позиции вставки встречается заполненный узел, ситуация усложняется. Такие узлы должны *разбиваться*; именно процесс разбиения сохраняет сбалансированность дерева. Разновидность деревьев 2-3-4, рассматриваемых нами, часто называется *нисходящими деревьями 2-3-4*, потому что узлы разбиваются в процессе перемещения вниз к позиции вставки.

Назовем элементы данных в разбиваемом узле A, B и C. Процесс разбиения происходит следующим образом (предполагается, что разбиваемый узел не является корневым; разбиение корневого узла будет рассмотрено позднее).

- Создается новый пустой узел. Он является «братом» (одноуровневым узлом) по отношению к разбиваемому узлу и размещается справа от него.
- Элемент данных C перемещается в новый узел.
- Элемент данных B перемещается в родителя разбиваемого узла.
- Элемент данных A остается на своем месте.
- Два правых потомка отсоединяются от разбиваемого узла и связываются с новым узлом.

Пример разбиения представлен на рис. 10.5. Разбиение также можно описать как преобразование 4-узла в два 2-узла.

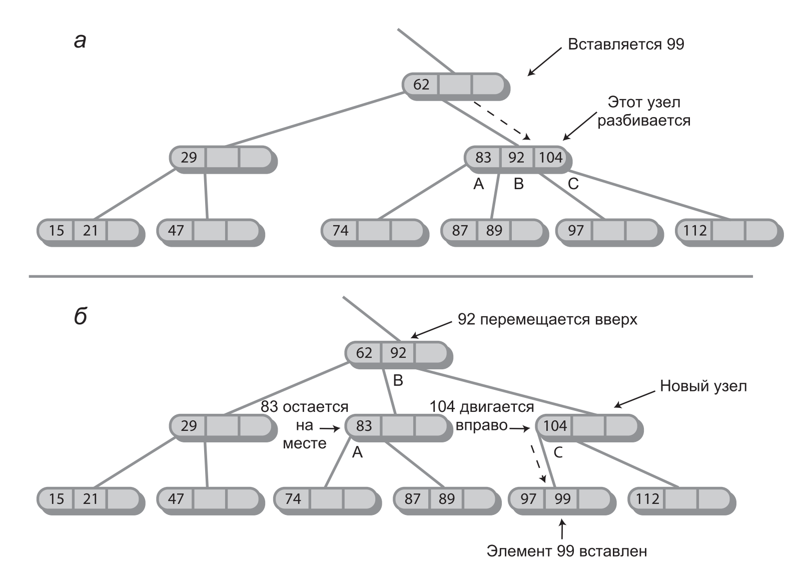


Рис. 10.5. Разбиение узла: *а* — до вставки; *б* — после вставки

Обратите внимание: в результате разбиения узла данные смещаются направо и вверх. Такое перемещение обеспечивает сбалансированность дерева.

В этом примере вставка требует разбиения только одного узла, но на пути к позиции вставки может быть обнаружено несколько полных узлов. В таких ситуациях приходится выполнять несколько разбиений.

Разбиение корневого узла

Если в самом начале поиска позиции вставки обнаруживается полный корневой узел, процесс разбиения несколько усложняется:

- ✦ Создается новый корневой узел, он становится родителем разбиваемого узла.
- ✦ Создается второй новый узел, который становится «братом» разбиваемого узла.
- ✦ Элемент данных C перемещается в созданного «брата».
- ✦ Элемент данных B перемещается в созданный корневой узел.
- ✦ Элемент данных A остается на своем месте.
- ✦ Два правых потомка отсоединяются от разбиваемого узла и связываются с новым правым узлом.

На рис. 10.6 показано, как происходит разбиение корневого узла. Процесс создает новый корень, находящийся уровнем выше старого. Таким образом, общая высота дерева увеличивается на единицу. Разбиение корня также можно описать как преобразование 4-узла в три 2-узла.

После разбиения узла поиск позиции вставки продолжается вниз по дереву. На рис. 10.6 элемент данных с ключом 41 вставляется в соответствующий узел.

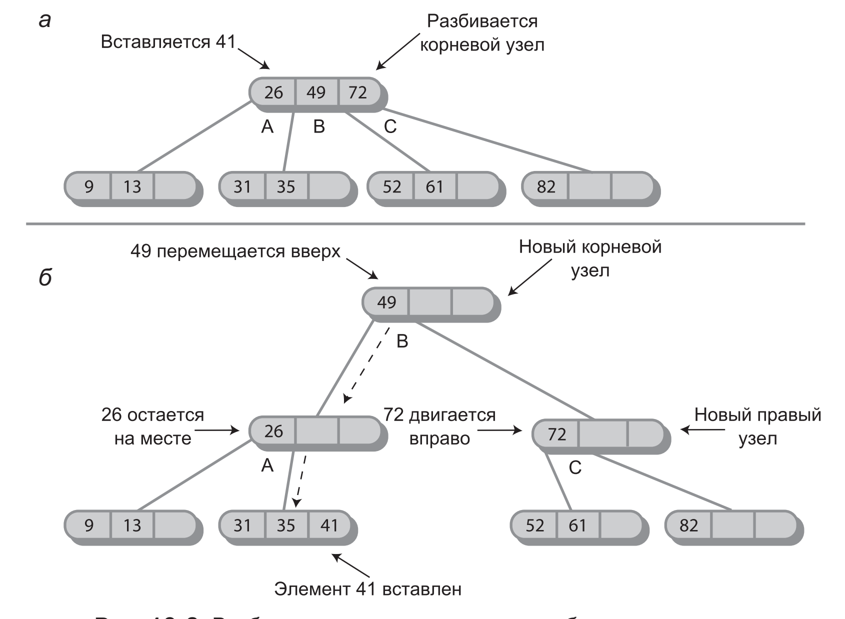


Рис. 10.6. Разбиение корня: *а* — до вставки; *б* — после вставки

Разбиение при перемещении вниз

Так как все полные узлы разбиваются при перемещении вниз, разбиение не может привести к каскадному распространению последствий по дереву. Родитель любого разбиваемого узла заведомо не является полным, а следовательно, может принять элемент данных B без необходимости разбиения. Конечно, если родитель уже имеет двух потомков на момент разбиения его потомка, он станет полным узлом, однако это означает лишь то, что он будет разбит при следующем поиске.

На рис. 10.7 показана серия вставок в пустое дерево. Всего выполняются четыре разбиения: два для корневого узла и два для листьев.

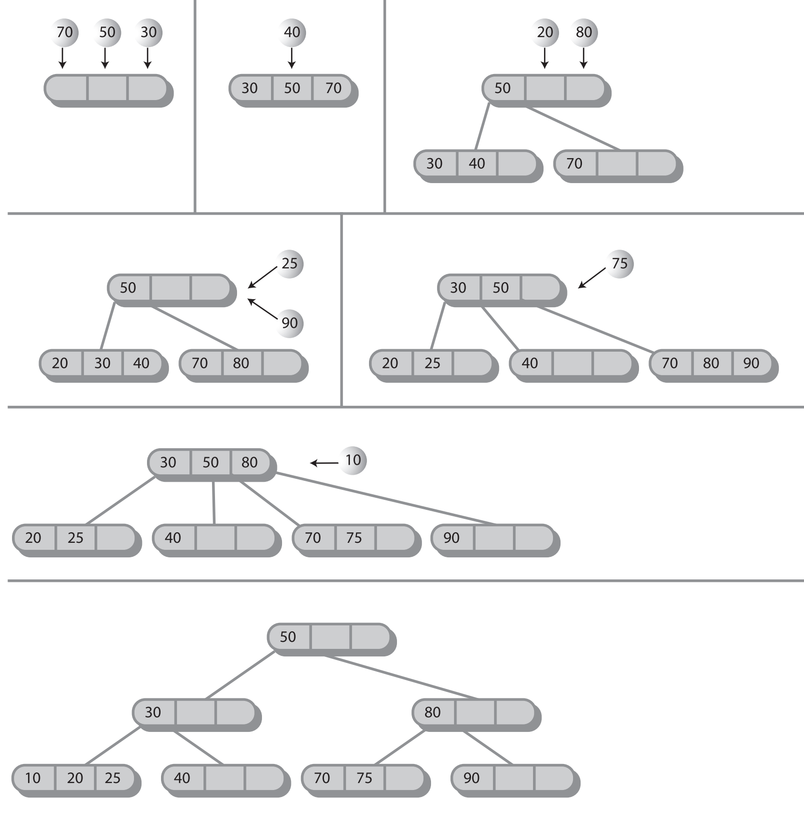


Рис. 10.7. Вставка в дерево 2-3-4

Приложение Tree234 Workshop

Приложение Tree234 Workshop поможет разобраться в том, как работают деревья 2-3-4. На рис. 10.8 показано, как выглядит рабочая область приложения при запуске.

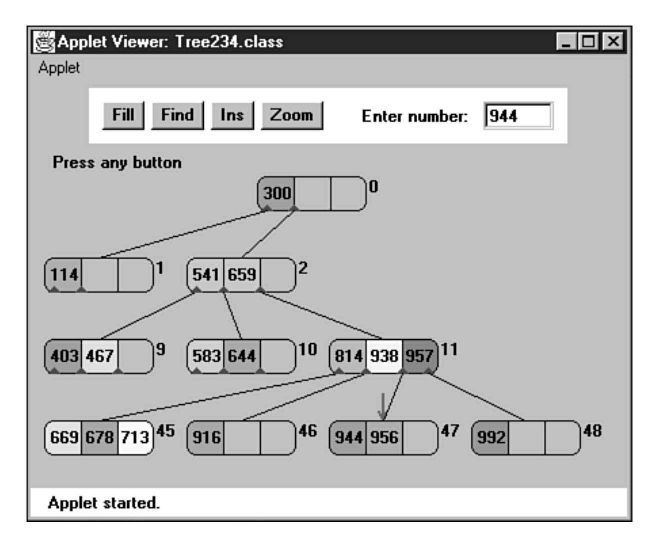


Рис. 10.8. Приложение Tree234 Workshop

Кнопка Fill

При запуске приложение Tree234 Workshop вставляет в дерево 7 элементов данных. Кнопка `Fill` позволяет создать новое дерево с другим количеством элементов данных (от 0 до 45). Щелкните на кнопке `Fill` и введите число в текстовом поле. Следующий щелчок создает новое дерево.

Дерево с 45 узлами выглядит не очень полным, но увеличение числа узлов потребует большего количества уровней, которые не поместятся в приложении.

Кнопка Find

Чтобы проследить за тем, как приложение ищет элемент данных с заданным ключом, многократно нажимайте кнопку `Find`. По запросу приложения введите искомый ключ, а затем при следующих нажатиях кнопки понаблюдайте за перемещениями красной стрелки от узла к узлу.

В приложении выводятся сообщения вида *Went to child number 1* (Переход к потомку 1). Как вы уже знаете, потомки нумеруются от 0 до 3 слева направо, а элементы данных нумеруются от 0 до 2. После непродолжительной тренировки вы научитесь прогнозировать путь, по которому пойдет поиск.

Поиск требует проверки одного узла на каждом уровне. Приложение поддерживает до четырех уровней, поэтому любой элемент находится проверкой не более четырех узлов. В каждом не-листовом узле алгоритм проверяет каждый элемент данных, начиная слева; так он определяет, совпадает ли элемент с искомым ключом, а если нет — к какому потомку следует перейти. В листовом узле на совпадение с искомым ключом проверяются все элементы данных. Если ключ отсутствует в листовом узле, поиск завершается неудачей.

В приложении Tree234 Workshop каждая операция должна быть завершена до начала следующей операции. Щелкайте на кнопке, пока не появится сообщение *Press any button* — признак завершения операции.

Кнопка Ins

Кнопка Ins вставляет в дерево новый элемент данных с ключом, введенным в текстовом поле. Алгоритм сначала ищет подходящий узел. Если на пути поиска будет обнаружен полный узел, он разбивается, после чего поиск продолжается.

Поэкспериментируйте с процессом вставки. Проследите за тем, что происходит при отсутствии полных узлов на пути к позиции вставки; процесс весьма тривиален. Затем попытайтесь вставить элемент в конце пути, включающего полный узел (в корне, в листе или где-то посередине). Проследите за тем, как образуются новые узлы, а содержимое разбиваемого узла распределяется между другими узлами.

Кнопка Zoom

В деревьях 2-3-4 количество узлов и элементов данных стремительно растет от уровня к уровню. Приложение Tree234 Workshop поддерживает всего 4 уровня, но теоретически нижний уровень может содержать 64 узла, каждый из которых вмещает до трех элементов данных.

Вывести такое количества элементов в одном ряду нереально, поэтому приложение выводит только часть из них: потомков выделенного узла. (Чтобы увидеть потомков другого узла, достаточно щелкнуть на нем; вскоре мы вернемся к этой теме.) Чтобы увидеть все дерево в уменьшенном виде, щелкните на кнопке Zoom (рис. 10.9).

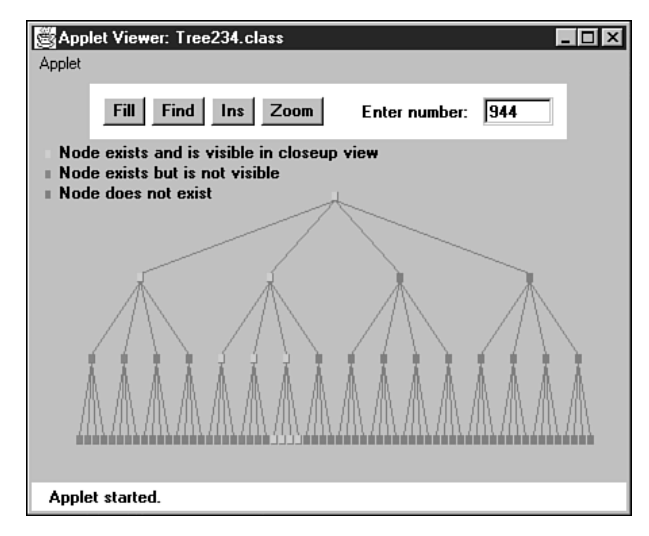


Рис. 10.9. Дерево 2-3-4 в уменьшении

В этом представлении узлы отображаются в виде маленьких прямоугольников, а элементы данных не показаны. Узлы, существующие и видимые в настоящее время в нормальном режиме (который можно восстановить повторным нажатием кнопки Zoom), окрашены в зеленый цвет. Узлы существующие, но невидимые в нормальном режиме, окрашены в малиновый цвет, а несуществующие узлы — в серый. На печатной странице различить цвета нелегко, зато на цветном мониторе все очевидно.

Переключение между уменьшенным и нормальным видом при помощи кнопки Zoom позволяет увидеть как общую картину, так и подробности — и хочется надеяться, связать их в вашем воображении.

Просмотр разных узлов

В нормальном режиме всегда видны все узлы двух верхних рядов: единственный корневой узел в верхнем ряду и четыре узла во втором. Рассмотреть следующий ряд уже проблематично, потому что количество узлов выходит за рамки рабочей области приложения: 16 в третьем ряду, 64 в четвертом. Однако вы можете просмотреть любой узел, щелкнув на его родителе (или иногда сначала на предке, а потом на родителе).

Синий треугольник у низа узла показывает, где потомок соединяется с узлом. Если потомки узла видны в настоящий момент, то связи с потомками отображаются в виде линий, идущих к ним от синих треугольников. Если потомки не видны, то линий нет, но синие треугольники все равно отображаются — они показывают, что у узла есть потомки. Если щелкнуть на родительском узле, в приложении отображаются его потомки и идущие к ним линии. Щелкая на соответствующих узлах, пользователь перемещается по всему дереву.

Для удобства все узлы пронумерованы от 0 (корень) до 85 (крайний правый узел в нижнем ряду). Номера отображаются в правом верхнем углу узлов (см. рис. 10.8). Нумерация не зависит от того, существует или нет узел в некоторой позиции, поэтому номера существующих узлов могут не образовывать непрерывную последовательность.

На рис. 10.10 изображено небольшое дерево с четырьмя узлами в третьем ряду. Пользователь щелкнул на узле 1, поэтому в приложении видны два его потомка с номерами 5 и 6. Если пользователь щелкнет на узле 2, то появятся два его потомка с номерами 9 и 10 (рис. 10.11).

Иллюстрации показывают, как можно переключаться между разными узлами третьего ряда, щелкая на узлах второго ряда. Чтобы изменить узлы в четвертом ряду, необходимо сначала щелкнуть на предке во втором ряду, а затем на родителе в третьем ряду.

При выполнении поиска и вставки кнопками Find и Ins представление автоматически переключается на узел, на который в данный момент указывает красная стрелка.

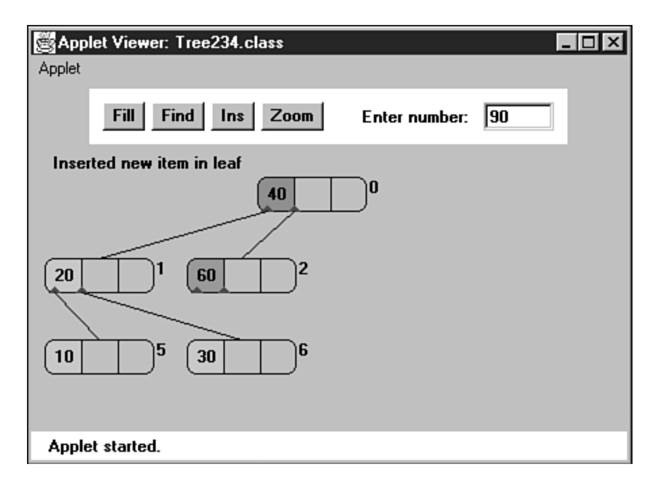


Рис. 10.10. Выбор левых потомков

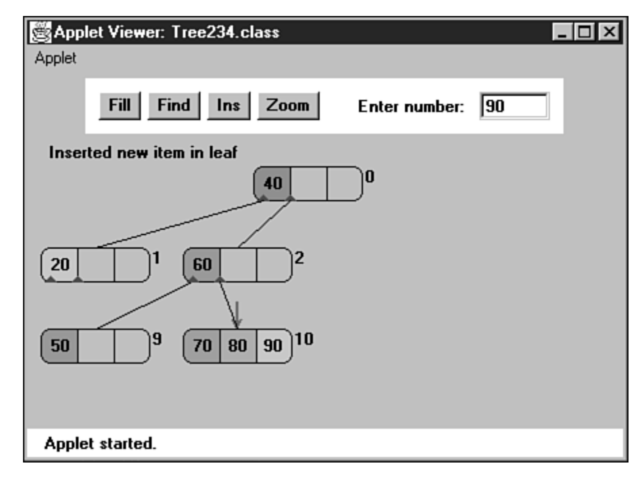


Рис. 10.11. Выбор правых потомков

Эксперименты

Приложение Tree234 Workshop позволяет быстро изучить принципы работы деревьев 2-3-4. Проследите за тем, как выполняется разбиение узлов. Сделайте паузу перед выполнением операции и попробуйте определить, куда переместятся три элемента данных из разбиваемого узла. Затем снова нажмите Ins и проверьте свои предположения.

С увеличением размеров дерева вам придется перемещаться по нему, чтобы увидеть все узлы. Щелкните на узле, чтобы просмотреть его потомков (а также потомков его потомков и т. д.). Если вы забудете, какая часть дерева просматривается в данный момент, щелкните на кнопке Zoom, чтобы увидеть дерево в уменьшенном виде.

Сколько элементов данных можно вставить в дерево? Их количество ограничено, потому что дерево может содержать не более четырех уровней. На четырех уровнях теоретически может размещаться 1 + 4 + 16 + 64, итого 85 узлов (все 85 узлов будут видны в уменьшенном виде). Если бы каждый узел содержал все три элемента данных, то в дереве помещалось бы 255 элементов. Тем не менее все узлы не могут быть полными одновременно. Задолго до их заполнения очередное разбиение корневого узла потребует создания пятого уровня, а это невозможно, потому что приложение поддерживает только четыре уровня. Теоретически элементы можно намеренно вставлять в узлы на путях, не содержащих полных узлов, чтобы избежать разбиений; это позволит заполнить бóльшую часть элементов данных — но для реальных данных такое решение, конечно, неприемлемо. Для случайных данных вряд ли можно рассчитывать на вставку в приложении более 50 элементов. Кнопка `Fill` ограничивает их количество 45, чтобы свести к минимуму риск переполнения.

Реализация дерева 2-3-4 на языке Java

В этом разделе рассматривается Java-программа, моделирующая дерево 2-3-4. Полный код программы `tree234.java` приведен в конце раздела. Программа довольно сложная, а ее классы тесно связаны друг с другом; чтобы понять, как она работает, придется внимательно просмотреть весь листинг.

Программа состоит из четырех классов: `DataItem`, `Node`, `Tree234` и `Tree234App`. Сейчас мы поочередно рассмотрим все эти классы.

Класс DataItem

Объекты класса `DataItem` представляют элементы данных, хранящиеся в узлах. В реальных программах каждый объект обычно содержит целую запись с информацией о работнике, детали машины и т. д., но в нашем случае с каждым объектом `DataItem` связан только один элемент данных типа `long`. Объекты этого класса умеют выполнять только два действия: инициализироваться и выводить свое текущее значение. Перед значением выводится косая черта: `/27`. (Последний метод вызывается методом `display` класса `Node` для вывода всех элементов в узле.)

Класс Node

Класс `Node` содержит два массива: `childArray` и `itemArray`. Первый массив состоит из четырех ячеек; в нем хранятся ссылки на потомков узла. Второй массив состоит из трех ячеек и содержит ссылки на объекты элементов данных `DataItem`, хранящиеся в узле.

Обратите внимание: элементы данных в `itemArray` образуют упорядоченный массив. Добавление новых и удаление существующих элементов производится по

тем же принципам, что и в любом упорядоченном массиве (см. главу 2, «Массивы»). Вставка нового элемента в нужной позиции может потребовать сдвига существующих элементов, а удаление — заполнения пустой ячейки.

В нашей реализации текущее количество элементов данных в узле (numItems) и ссылка на родителя узла (parent) хранятся в полях класса. Ни одно из этих полей не является абсолютно необходимым; их можно удалить, чтобы узлы занимали меньше места. Однако наличие этих полей делает код более наглядным, а это обстоятельство вполне оправдывает небольшое увеличение узлов.

Класс Node содержит разнообразные вспомогательные методы для управления связями с потомками и родителем, проверки полноты узла и того, является ли он листовым. И все же основная работа выполняется методами findItem(), insertItem() и removeItem(), которые работают с отдельными элементами данных в узлах. Эти методы предназначены соответственно для поиска узла, содержащего элемент данных с заданным ключом; вставки нового элемента данных в узел (с перемещением существующих элементов при необходимости); и удаления узла (также с перемещением существующих элементов при необходимости). Не путайте эти методы с методами find() и insert() класса Tree234, который будет рассматриваться следующим.

Метод display выводит содержимое узла с разделением элементов данных символом «/» — /27/56/89/, /14/66/ или /45/.

Так как в Java ссылки автоматически инициализируются значениями null, а числа обнуляются при создании объектов, классу Node конструктор не нужен.

Класс Tree234

Объект класса Tree234 представляет все дерево. Класс содержит всего одно поле root типа Node. Выполнение всех операций начинается с корневого узла, поэтому хранить какую-либо дополнительную информацию в объекте дерева не обязательно.

Поиск

Поиск элемента данных с заданным ключом выполняется методом find(). Метод начинает с корня дерева, и для каждого объекта узла вызывает метод findItem() этого объекта, чтобы узнать, находится ли элемент в узле. Если проверка дает положительный результат, метод возвращает индекс элемента в массиве элементов узла. Если узел find() является листовым и элемент в нем отсутствует, поиск завершается неудачей, поэтому метод возвращает –1. Если элемент не найден в текущем узле, а текущий узел не является листовым, find() вызывает метод getNextChild(), который определяет, к какому из потомков текущего узла следует перейти для продолжения поиска.

Вставка

Метод insert() начинается с блока кода, аналогичного find() — если не считать того, что обнаруженные полные узлы подвергаются разбиению. Кроме того, предполагается, что вставка всегда завершается успехом; метод продолжает искать, опу-

скаясь все глубже и глубже, пока не найдет листовой узел. Новый элемент данных вставляется в этот узел. (Свободное место в листе заведомо имеется, в противном случае он подвергся бы разбиению.)

Разбиение

Метод `split()` — самый сложный в этой программе. Узел, с которым необходимо выполнить разбиение, передается ему в аргументе. Сначала два правых элемента данных извлекаются из узла и сохраняются во временных переменных. Затем два правых потомка отсоединяются от узла; ссылки на них тоже временно сохраняются.

Метод создает новый узел с именем `newRight`. Созданный узел будет размещен справа от разбиваемого узла. Если разбиваемый узел является корневым, то метод создает дополнительный новый узел: корневой.

Далее устанавливаются связи с родителем разбиваемого узла. Им может быть уже существующий родитель, или в случае разбиения корневого узла — только что созданный корневой узел. Обозначим три элемента данных в разбиваемом узле буквами A, B и C. Элемент B вставляется в родительский узел. При необходимости существующие потомки родителя отсоединяются от него и присоединяются заново на одну позицию правее, чтобы освободить место для нового элемента данных и новых связей. После этого узел `newRight` присоединяется к родителю (см. рис. 10.5 и 10.6).

Теперь на передний план выходит узел `newRight`. В него вставляется элемент данных C, и к нему присоединяются потомки 2 и 3, которые ранее были отсоединены от разбиваемого узла. На этом разбиение завершается, а метод `split()` возвращает управление.

Класс Tree234App

В классе `Tree234App` метод `main()` вставляет в дерево несколько элементов данных. Он предоставляет в распоряжение пользователя текстовый интерфейс с командами *s* (просмотр дерева), *i* (вставка нового элемента данных) и *f* (поиск существующего элемента). Сеанс взаимодействия с программой выглядит примерно так:

```
Enter first letter of show, insert, or find: s
level=0 child=0 /50/
level=1 child=0 /30/40/
level=1 child=1 /60/70/

Enter first letter of show, insert, or find: f
Enter value to find: 40
Found 40

Enter first letter of show, insert, or find: i
Enter value to insert: 20
Enter first letter of show, insert, or find: s
level=0 child=0 /50/
```

```
level=1 child=0 /20/30/40/
level=1 child=1 /60/70/

Enter first letter of show, insert, or find: i
Enter value to insert: 10
Enter first letter of show, insert, or find: s
level=0 child=0 /30/50/
level=1 child=0 /10/20/
level=1 child=1 /40/
level=1 child=2 /60/70/
```

Выходные данные не слишком наглядны, но при желании вы можете сами нарисовать дерево по этим данным. В содержимом дерева выводится уровень (0 соответствует корневому) и номер потомка. Алгоритм вывода выполняет перебор в глубину, поэтому сначала выводится корневой узел, затем его первый потомок и поддерево, корнем которого он является, далее второй потомок со своим поддеревом и т. д.

В выходных данных видны два вставленных элемента: 20 и 10. Вставка второго из них привела к разбиению узла (потомка 0 корневого узла). На рис. 10.12 изображено дерево, образовавшееся в результате этих вставок после завершающего нажатия клавиши *s*.

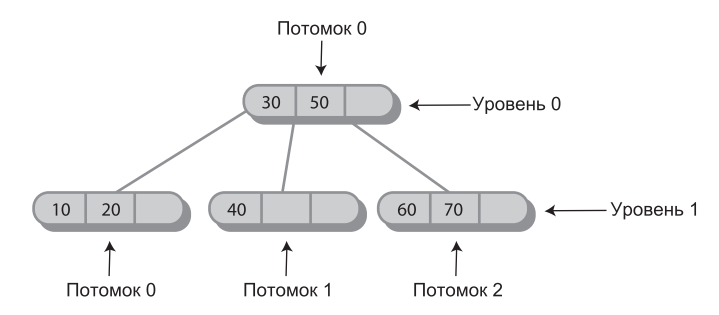


Рис. 10.12. Пример вывода программы tree234.java

Полный код программы tree234.java

В листинге 10.1 представлен полный код программы `tree234.java` со всеми упомянутыми классами. Как и в большинстве объектно-ориентированных программ, проще начать с изучения высокоуровневых классов и постепенно переходить к классам, реализующим низкоуровневые подробности. В этой программе такой подход соответствует порядку рассмотрения `Tree234App`, `Tree234`, `Node`, `DataItem`.

Листинг 10.1. Программа tree234.java

```
// tree234.java
// Работа с 234-деревом
// Запуск программы: C>java Tree234App
import java.io.*;
////////////////////////////////////////////////////////////////
```

```
class DataItem
   {
   public long dData;          // Один объект данных
//--------------------------------------------------------------
   public DataItem(long dd)    // Конструктор
      { dData = dd; }
//--------------------------------------------------------------
   public void displayItem()   // Вывод элемента в формате "/27"
      { System.out.print("/"+dData); }
//--------------------------------------------------------------
   }  // Конец класса DataItem
////////////////////////////////////////////////////////////////
class Node
   {
   private static final int ORDER = 4;
   private int numItems;
   private Node parent;
   private Node childArray[] = new Node[ORDER];
   private DataItem itemArray[] = new DataItem[ORDER-1];
// -------------------------------------------------------------
   // Связывание узла с потомком
   public void connectChild(int childNum, Node child)
      {
      childArray[childNum] = child;
      if(child != null)
         child.parent = this;
      }
// -------------------------------------------------------------
   // Метод отсоединяет потомка от узла и возвращает его
   public Node disconnectChild(int childNum)
      {
      Node tempNode = childArray[childNum];
      childArray[childNum] = null;
      return tempNode;
      }
// -------------------------------------------------------------
   public Node getChild(int childNum)
      { return childArray[childNum]; }
// -------------------------------------------------------------
   public Node getParent()
      { return parent; }
// -------------------------------------------------------------
   public boolean isLeaf()
      { return (childArray[0]==null) ? true : false; }
// -------------------------------------------------------------
   public int getNumItems()
      { return numItems; }
// -------------------------------------------------------------
   public DataItem getItem(int index)   // Получение объекта DataItem
      { return itemArray[index]; }      // с заданным индексом
```

продолжение ⇨

Листинг 10.1 (*продолжение*)

```
// -------------------------------------------------------------
   public boolean isFull()
      { return (numItems==ORDER-1) ? true : false; }
// -------------------------------------------------------------
   public int findItem(long key)       // Определение индекса элемента
      {                                   // (в пределах узла)
      for(int j=0; j<ORDER-1; j++)        // Если элемент найден
         {                                // В противном случае
         if(itemArray[j] == null)         // Вернуть -1
            break;
         else if(itemArray[j].dData == key)
            return j;
         }
      return -1;
      }
// -------------------------------------------------------------
   public int insertItem(DataItem newItem)
      {
      // Предполагается, что узел не пуст
      numItems++;                         // Добавление нового элемента
      long newKey = newItem.dData;        // Ключ нового элемента

      for(int j=ORDER-2; j>=0; j--)       // Начиная справа,
         {                                //    проверяем элементы
         if(itemArray[j] == null)         // Если ячейка пуста,
            continue;                     // перейти на одну ячейку влево
         else                             // Если нет,
            {                             // получить ее ключ
            long itsKey = itemArray[j].dData;
            if(newKey < itsKey)           // Если новый ключ больше
               itemArray[j+1] = itemArray[j]; // Сдвинуть вправо
            else
               {
               itemArray[j+1] = newItem;  // Вставка нового элемента
               return j+1;                // Метод возвращает индекс
               }                          // нового элемента
            }
         }                                // Все элементы сдвинуты,
      itemArray[0] = newItem;             // вставка нового элемента
      return 0;
      }
// -------------------------------------------------------------
   public DataItem removeItem()        // Удаление наибольшего
      {                                // элемента
      // Предполагается, что узел не пуст
      DataItem temp = itemArray[numItems-1];  // Сохранение элемента
      itemArray[numItems-1] = null;           // Отсоединение
      numItems--;                             // На один элемент меньше
      return temp;                            // Метод возвращает
      }                                       // удаленный элемент
```

```
// -------------------------------------------------------------
   public void displayNode()           // Формат "/24/56/74/"
      {
      for(int j=0; j<numItems; j++)
         itemArray[j].displayItem();   // "/56"
      System.out.println("/");         // Завершающий символ "/"
      }
// -------------------------------------------------------------
   }  // Конец класса Node
////////////////////////////////////////////////////////////////
class Tree234
   {
   private Node root = new Node();            // Создание корневого узла
// -------------------------------------------------------------
   public int find(long key)
      {
      Node curNode = root;
      int childNumber;
      while(true)
         {
         if(( childNumber=curNode.findItem(key) ) != -1)
            return childNumber;               // Узел найден
         else if( curNode.isLeaf() )
            return -1;                        // Узел не найден
         else                                 // Искать глубже
            curNode = getNextChild(curNode, key);
         }
      }
// -------------------------------------------------------------
   // Вставка элемента данных
   public void insert(long dValue)
      {
      Node curNode = root;
      DataItem tempItem = new DataItem(dValue);

      while(true)
         {
         if( curNode.isFull() )               // Если узел полон,
            {
            split(curNode);                   // он разбивается.
            curNode = curNode.getParent();    // Возврат уровнем выше
                                              // Поиск
            curNode = getNextChild(curNode, dValue);
            }

         else if( curNode.isLeaf() )          // Если узел листовой,
            break;                            // переход к вставке
         // Узел не полный и не листовой; спуститься уровнем ниже
         else
            curNode = getNextChild(curNode, dValue);
         }
```

продолжение ⇗

Листинг 10.1 (*продолжение*)

```
      curNode.insertItem(tempItem);       // Вставка нового объекта DataItem
      }
// -------------------------------------------------------------
   public void split(Node thisNode)     // Разбиение узла
      {
      // Предполагается, что узел полон
      DataItem itemB, itemC;
      Node parent, child2, child3;
      int itemIndex;

      itemC = thisNode.removeItem();    // Удаление элементов из
      itemB = thisNode.removeItem();    // текущего узла
      child2 = thisNode.disconnectChild(2); // Отсоединение потомков
      child3 = thisNode.disconnectChild(3); // от текущего узла
      Node newRight = new Node();       // Создание нового узла

      if(thisNode==root)                // Если узел является корнем,
         {
         root = new Node();                // Создание нового корня
         parent = root;                    // Корень становится родителем
         root.connectChild(0, thisNode);   // Связывание с родителем
         }
      else                              // Текущий узел не является корнем
         parent = thisNode.getParent();    // Получение родителя

      // Разбираемся с родителем
      itemIndex = parent.insertItem(itemB); // B вставляется в родителя
      int n = parent.getNumItems();         // Всего элементов?

      for(int j=n-1; j>itemIndex; j--)          // Перемещение связей
         {                                      // родителя
         Node temp = parent.disconnectChild(j); // На одного потомка
         parent.connectChild(j+1, temp);        // вправо
         }
                                  // Связывание newRight с родителем
      parent.connectChild(itemIndex+1, newRight);

      // Разбираемся с узлом newRight
      newRight.insertItem(itemC);       // Элемент C в newRight
      newRight.connectChild(0, child2); // Связывание 0 и 1
      newRight.connectChild(1, child3); // с newRight
      }
// -------------------------------------------------------------
   // Получение соответствующего потомка при поиске значения
   public Node getNextChild(Node theNode, long theValue)
      {
      int j;
      // Предполагается, что узел не пуст, не полон и не является листом
      int numItems = theNode.getNumItems();
```

```
      for(j=0; j<numItems; j++)          // Для каждого элемента в узле
         {                               // Наше значение меньше?
         if( theValue < theNode.getItem(j).dData )
            return theNode.getChild(j);  // Вернуть левого потомка
         }                               // Наше значение больше,
      return theNode.getChild(j);        // Вернуть правого потомка
      }
// -------------------------------------------------------------
   public void displayTree()
      {
      recDisplayTree(root, 0, 0);
      }
// -------------------------------------------------------------
   private void recDisplayTree(Node thisNode, int level,
                                             int childNumber)
   {
   System.out.print("level="+level+" child="+childNumber+" ");
   thisNode.displayNode();               // Вывод содержимого узла

   // Рекурсивный вызов для каждого потомка текущего узла
   int numItems = thisNode.getNumItems();
   for(int j=0; j<numItems+1; j++)
      {
      Node nextNode = thisNode.getChild(j);
      if(nextNode != null)
         recDisplayTree(nextNode, level+1, j);
      else
         return;
      }
   }
// -------------------------------------------------------------
}  // Конец класса Tree234
////////////////////////////////////////////////////////////////
class Tree234App
   {
   public static void main(String[] args) throws IOException
      {
      long value;
      Tree234 theTree = new Tree234();

      theTree.insert(50);
      theTree.insert(40);
      theTree.insert(60);
      theTree.insert(30);
      theTree.insert(70);
      while(true)
         {
         System.out.print("Enter first letter of ");
         System.out.print("show, insert, or find: ");
```

продолжение ⇨

Листинг 10.1 (*продолжение*)

```
        char choice = getChar();
        switch(choice)
           {
           case 's':
              theTree.displayTree();
              break;
           case 'i':
              System.out.print("Enter value to insert: ");
              value = getInt();
              theTree.insert(value);
              break;
           case 'f':
              System.out.print("Enter value to find: ");
              value = getInt();
              int found = theTree.find(value);
              if(found != -1)
                 System.out.println("Found "+value);
              else
                 System.out.println("Could not find "+value);
                 break;
              default:
                 System.out.print("Invalid entry\n");
              }
           }
        }
//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   public static char getChar() throws IOException
      {
      String s = getString();
      return s.charAt(0);
      }
//-------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
//-------------------------------------------------------------
   }  // Конец класса Tree234App
////////////////////////////////////////////////////////////////
```

Деревья 2-3-4 и красно-черные деревья

Вероятно, к этому моменту деревья 2-3-4 и красно-черные деревья (см. главу 9) кажутся вам чем-то разными конструкциями. Однако, как выясняется, в определенном смысле они полностью эквивалентны. Одно дерево может быть преобразовано в другое по нескольким несложным правилам, и даже операции, необходимые для сохранения их сбалансированности, эквивалентны. Математик скажет, что эти разновидности деревьев *изоморфны*.

Возможно, на практике вам не понадобится преобразовывать деревья 2-3-4 в красно-черные деревья, но эквивалентность этих структур углубит ваше понимание их работы, а также принесет пользу при анализе эффективности.

Сначала были изобретены деревья 2-3-4; позднее на их основе была разработана концепция красно-черных деревьев.

Преобразование деревьев 2-3-4 в красно-черные деревья

Преобразование дерева 2-3-4 в красно-черное дерево определяется тремя правилами (рис. 10.13):

- Каждый 2-узел дерева 2-3-4 преобразуется в черный узел красно-черного дерева (рис. 10.13, *а*).
- Каждый 3-узел преобразуется в родительский узел с потомком (рис. 10.13, *б*). Потомок также связан с двумя потомками: либо W и X, либо X и Y, при этом родитель связан с другим потомком Y или W. Неважно, какой узел становится родителем, а какой — потомком. Потомок окрашивается в красный цвет, а родитель в черный.
- Каждый 4-узел преобразуется в родителя с двумя потомками (рис. 10.13, *в*). Первый потомок связывается с двумя своими потомками W и X, а второй — с потомками Y и Z. Как и в предыдущем случае, потомки окрашиваются в красный цвет, а родитель в черный.

На рис. 10.14 показано дерево 2-3-4 и соответствующее красно-черное дерево, полученное в результате применения этих преобразований. Пунктирными линиями обозначены поддеревья, созданные на основе 3-узлов и 4-узлов. Красно-черные правила автоматически выполняются в построенном дереве. Убедитесь в том, что это действительно так: два красных узла никогда не связываются отношениями «родитель/потомок», а все пути от корня к листовому узлу (или пустому потомку) содержат одинаковое количество черных узлов. Можно сказать, что 3-узел дерева 2-3-4 эквивалентен родителю с красным потомком в красно-черном дереве, а 4-узел эквивалентен родителю с двумя красными потомками. Черный родитель с черным потомком в красно-черном дереве не соответствует 3-узлу в дереве 2-3-4; он соответствует 2-узлу с другим 2-узлом в качестве потомка. Аналогичным образом черный родитель с двумя черными потомками не соответствует 4-узлу.

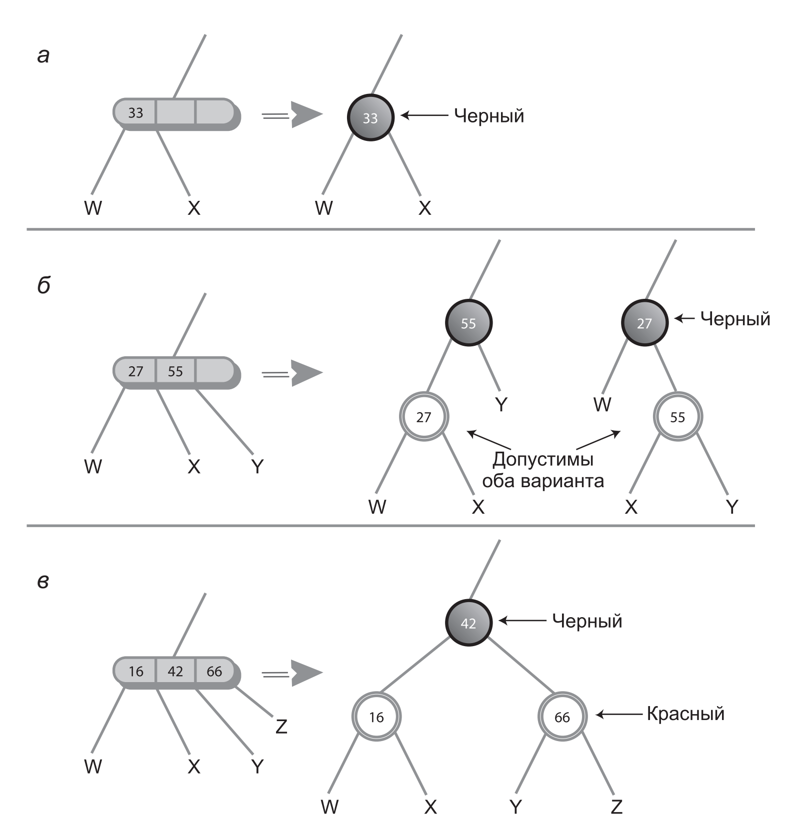


Рис. 10.13. Преобразование дерева 2-3-4 в красно-черное дерево

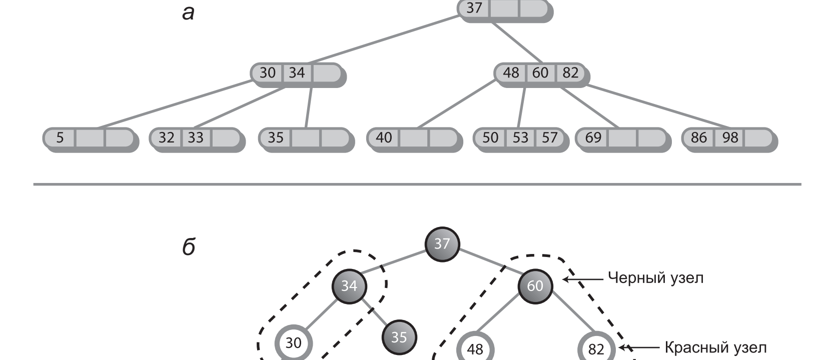


Рис. 10.14. Дерево 2-3-4 (*а*) и аналогичное ему красно-черное дерево (*б*)

Эквивалентность операций

Не только структура красно-черного дерева отображается на структуру дерева 2-3-4, но и операции, применяемые к этим двум разновидностям деревьев, тоже эквивалентны. Сбалансированность дерева 2-3-4 обеспечивается за счет разбиения узлов, а в красно-черном дереве для этой цели применяются переключения цветов и повороты.

Разбиения и повороты для 4-узлов

Спускаясь по дереву 2-3-4 в поисках позиции вставки для нового узла, мы разбиваем каждый 4-узел на два 2-узла. В красно-черном дереве применяются переключения цветов. Как связаны между собой эти операции?

На рис. 10.15, *а* изображен 4-узел дерева 2-3-4 перед разбиением; на рис. 10.15, *б* изображена ситуация после разбиения. 2-узел, который был родителем 4-узла, становится 3-узлом.

На рис. 10.15, *в* изображен красно-черный эквивалент дерева 2-3-4 на рис. 10.15, *а*. Пунктирными линиями выделен эквивалент 4-узла. В результате переключения цветов образуется красно-черное дерево, изображенное на рис. 10.15, *г*. Узлы 40 и 60 становятся черными, а узел 50 — красным. Таким образом, 50 и его родитель образуют эквивалент 3-узла; это тот самый 3-узел, который образуется в результате разбиения на рис. 10.15, *б*.

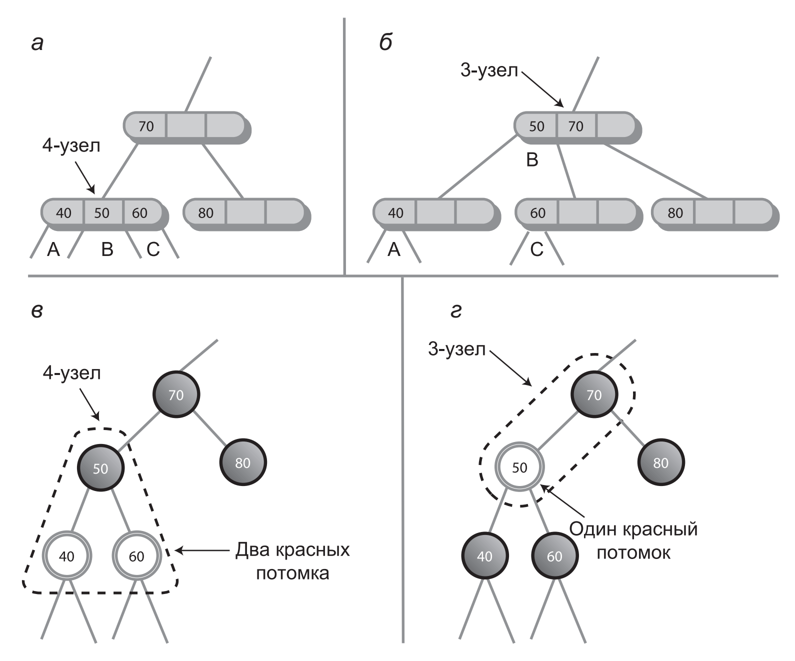


Рис. 10.15. Разбиение 4-узла и переключение цветов

Мы видим, что разбиение 4-узла в процессе вставки в дерево 2-3-4 эквивалентно переключению цветов в процессе вставки в красно-черное дерево.

Повороты и разбиения 3-узлов

При преобразовании 3-узла дерева 2-3-4 в красно-черный эквивалент возможны две конфигурации, как было показано ранее на рис. 10.13, *б*. Каждый из двух элементов данных может стать родителем. В зависимости от того, какой из элементов будет выбран, потомок становится левым или правым, а линия, соединяющая родителя с потомком, может быть наклонена в одну из двух сторон.

Обе конфигурации действительны, но они вносят разный вклад в сбалансированность дерева. Рассмотрим ситуацию в более широком контексте.

На рис. 10.16, *а* изображено дерево 2-3-4, а на рис. 10.16, *б* и *в* — эквивалентные красно-черные деревья, полученные в результате применения правил преобразования к дереву 2-3-4. Эти два дерева различаются выбором элемента данных 3-узла, который становится родителем: на рис. 10.16, *б* родителем становится элемент 80, а на рис. 10.16, *в* — элемент 70.

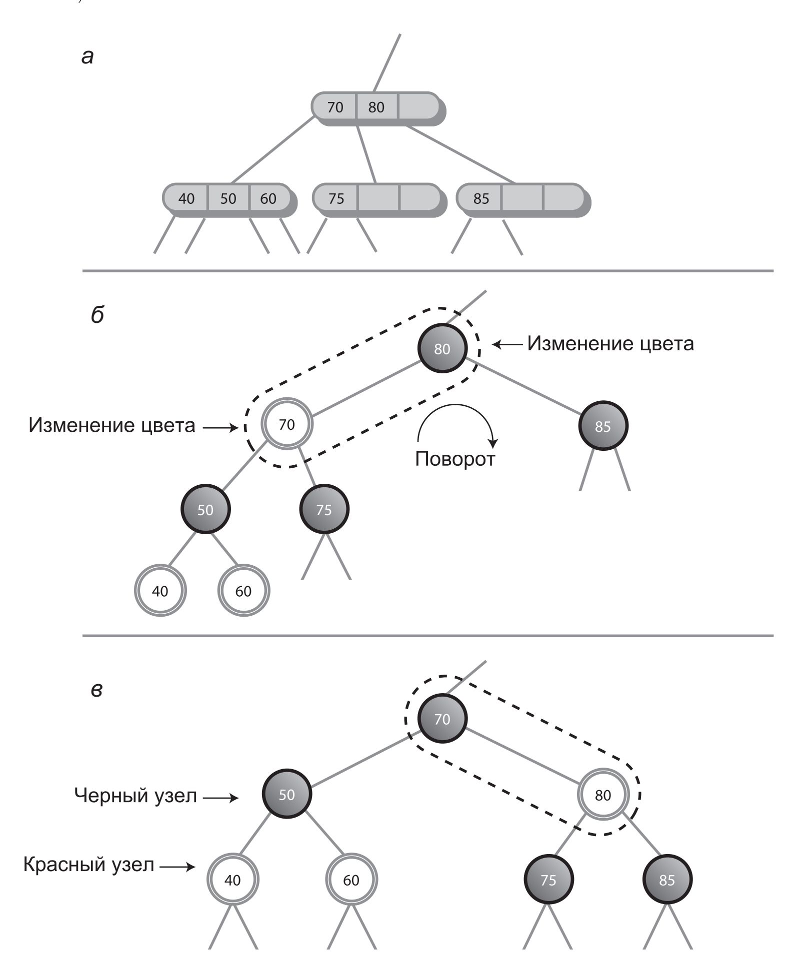


Рис. 10.16. 3-узлы и поворот: *а* — дерево 2-3-4; *б* — левый наклон; *в* — правый наклон

Хотя эти конфигурации в равной степени действительны, из рисунка видно, что случай *б* отличается от *в* несбалансированостью дерева. Чтобы сбалансировать красно-черное дерево на рис. 10.16, *б*, следует выполнить поворот направо (и выполнить два изменения цветов). Как ни удивительно, этот поворот приведет к дереву, изображенному на рис. 10.16, *в*.

Таким образом, мы видим, что между поворотами в красно-черных деревьях и выбором родительского узла при преобразовании дерева 2-3-4 в красно-черное дерево существует эквивалентность. И хотя здесь этот факт не продемонстрирован, аналогичная эквивалентность существует для двойных поворотов в конфигурации с внутренними внуками.

Эффективность деревьев 2-3-4

Оценка эффективности деревьев 2-3-4 несколько сложнее, чем у красно-черных деревьев, но эквивалентность красно-черных деревьев и деревьев 2-3-4 станет хорошей отправной точкой для анализа.

Скорость

Как было показано в главе 8, процесс поиска в красно-черном дереве требует посещения одного узла на каждом уровне (как при поиске существующих узлов, так и при вставке новых). Количество уровней в красно-черном дереве (сбалансированном двоичном дереве) составляет около $\log_2(N + 1)$, поэтому время поиска пропорционально этой величине.

Дерево 2-3-4 также требует посещения одного узла на каждом уровне, но дерево 2-3-4 короче (то есть содержит меньше уровней), чем красно-черное дерево с тем же количеством элементов данных. Например, на рис. 10.14 дерево 2-3-4 содержит три уровня, а красно-черное дерево — пять уровней.

Выражаясь конкретнее, в деревьях 2-3-4 узел может иметь до четырех потомков. Если бы все узлы дерева были заполнены, то его высота была бы пропорциональна $\log_4 N$. Логарифмы по основанию 2 и 4 отличаются постоянным коэффициентом 2. Таким образом, высота дерева 2-3-4 составляет около половины высоты красно-черного дерева — при условии, что все узлы дерева полны. Так как узлы не полны, высота дерева 2-3-4 лежит где-то в диапазоне от $\log_2(N + 1)$ до $\log_2(N + 1)/2$. Уменьшение высоты дерева 2-3-4 слегка сокращает время поиска по сравнению с красно-черными деревьями.

С другой стороны, каждый узел содержит больше элементов данных, а это увеличивает время поиска. Так как для проверки элементов данных в узлах используется линейный поиск, время поиска увеличивается на величину, пропорциональную M (среднее количество элементов на узел). Соответственно время поиска пропорционально $M \times \log_4 N$. В O-синтаксисе подобные малые постоянные множители игнорируются.

Итак, для деревьев 2-3-4 увеличение количества элементов на узел обычно компенсирует уменьшение высоты дерева. Времена поиска в дереве 2-3-4 и сбалансированных двоичных деревьях (таких, как красно-черные деревья) приблизительно равны, и в обоих случаях составляют $O(\log N)$.

Затраты памяти

В каждом узле дерева 2-3-4 резервируется память для хранения трех ссылок на элементы данных и четырех ссылок на потомков. Память может выделяться как в виде массивов (как в программе `tree234.java`), так и в виде отдельных переменных. Не вся память используется постоянно — в узле с одним элементом данных неэффективно тратится 2/3 памяти данных и 1/2 памяти потомков. Узел с двумя элементами данных неэффективно расходует 1/3 памяти данных и 1/4 памяти потомков; иначе говоря, он использует 5/7 доступного пространства. Если в среднем в каждом узле хранится два элемента данных, то около 2/7 доступной памяти расходуется непроизводительно.

Возможно, кто-то предложит хранить ссылки на потомков и элементы данных в связанных списках вместо массивов, но непроизводительные затраты на организацию связанного списка для хранения 3–4 элементов вряд ли оправдают такое решение.

Красно-черные деревья вследствие своей сбалансированности содержат меньшее количество узлов, имеющих только одного потомка, поэтому почти вся память для ссылок на потомков расходуется эффективно. Кроме того, каждый узел содержит максимально возможное количество элементов данных — один. Следовательно, красно-черные деревья превосходят деревья 2-3-4 в отношении эффективности использования памяти.

В языке Java вместо самих объектов хранятся ссылки на них, поэтому различия между деревьями 2-3-4 и красно-черными деревьями могут оказаться несущественными, а реализуются деревья 2-3-4 безусловно проще. Однако в других языках, в которых ссылки не используются подобным образом, различия в затратах памяти между красно-черными деревьями и деревьями 2-3-4 могут оказаться весьма значительными.

Деревья 2-3

Мы также в общих чертах рассмотрим деревья 2-3: во-первых, они важны с исторической точки зрения, а во-вторых, продолжают использоваться во многих приложениях. Кроме того, некоторые операции с деревьями 2-3 применимы и к B-деревьям, описанным в следующем разделе. Наконец, интересно посмотреть, как незначительное изменение количества потомков на узел приводит к серьезным изменениям в алгоритмах дерева.

Деревья 2-3 аналогичны деревьям 2-3-4, если не считать того, что в них (как нетрудно догадаться по названию) каждый узел содержит на один элемент данных

и на одного потомка меньше. Это была первая разновидность многопутевых деревьев, изобретенная Дж. Хопкрофтом в 1970-х годах. B-деревья (частным случаем которых являются деревья 2-3-4) появились только в 1972 году.

Деревья 2-3 имеют много общего с деревьями 2-3-4. Узлы могут содержать один или два элемента данных с нулем, одним, двумя или тремя потомками. В остальном конфигурация ключей родителей и потомков остается неизменной. Теоретически вставка элемента данных в узел дерева 2-3 упрощается, поскольку она требует меньшего количества сравнений и перемещений. Как и в деревьях 2-3-4, операции вставки выполняются только с листовыми узлами, а все листовые узлы находятся на нижнем уровне.

Разбиение узлов

Поиск существующих элементов данных выполняется точно так же, как в деревьях 2-3-4, не считая изменений в количестве элементов данных и потомков. Напрашивается предположение, что вставка в дереве 2-3 тоже выполняется аналогично, но в разбиениях существуют очень значительные различия.

В любом дереве в разбиении узла задействованы три элемента данных: один остается в разбиваемом узле, другой перемещается вправо в новый узел, а третий перемещается вверх в родительский узел. Полный узел в дереве 2-3-4 содержит три элемента данных, которые перемещаются в три места. Однако полный узел дерева 2-3 содержит только два элемента данных. Где взять третий? Необходимо использовать новый элемент — тот, который вставляется в дерево.

В дереве 2-3-4 новый элемент вставляется после выполнения всех разбиений, а в дереве 2-3 он должен участвовать в разбиении. Так как элемент вставляется в листовой узел, разбиения при перемещении вниз невозможны. Если листовой узел, в который вставляется новый элемент, не является полным, то новый элемент может быть вставлен немедленно, но полный листовой узел должен быть подвергнут разбиению. Его два элемента и новый элемент распределяются между тремя узлами: существующим, новым и родительским. Если родитель не полон, то операция завершается (после присоединения нового узла). Ситуация изображена на рис. 10.17.

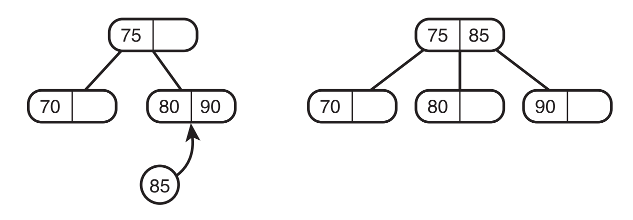


Рис. 10.17. Вставка с неполным родителем

Но если родительский узел полон, то его тоже необходимо разбить. Два его элемента и элемент, переданный от только что разбитого потомка, должны быть распределены между родителем, новым «братом» и родителем родителя (рис. 10.18).

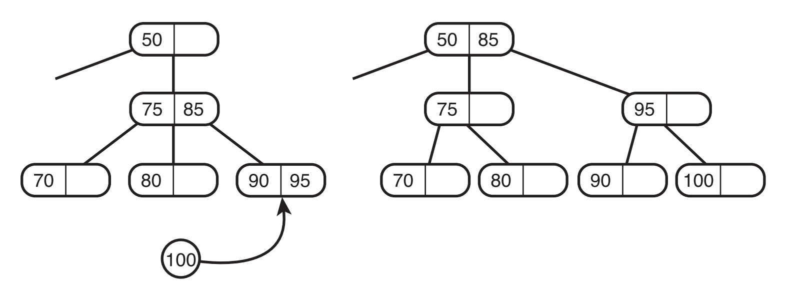


Рис. 10.18. Вставка с полным родителем

Если родитель родителя (предок листового узла) полон, то и его придется разбить. Процесс разбиения продвигается вверх до тех пор, пока на его пути не встретится неполный родитель или корневой узел. Если корневой узел полон, создается новый корень, который является родителем старого (рис. 10.19).

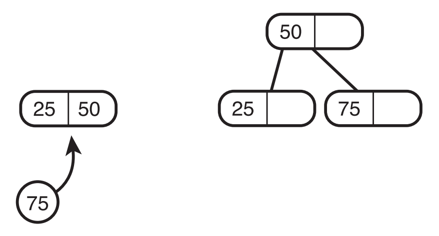


Рис. 10.19. Разбиение корневого узла

На рис. 10.20 показано, как разбиение по дереву распространяется до дереву вплоть до корневого узла.

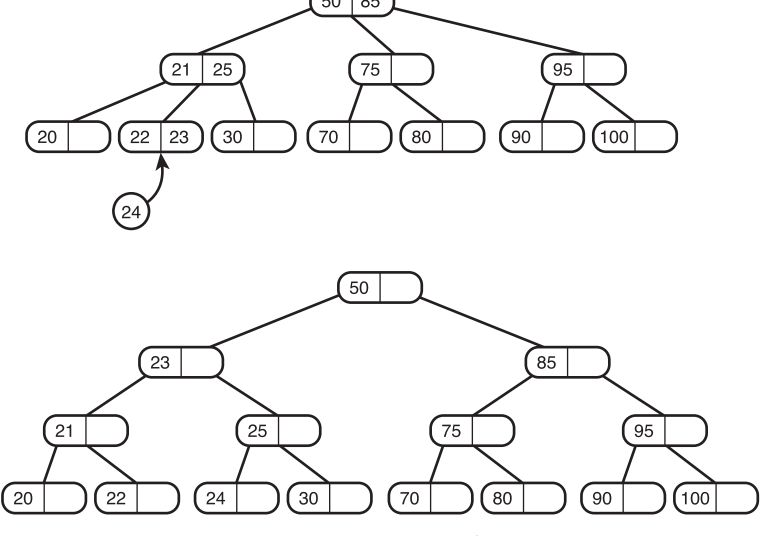


Рис. 10.20. Распространение разбиения по дереву

Реализация

Полная реализация дерева 2-3 на Java предоставляется читателю для самостоятельной работы. Тем не менее мы приведем некоторые рекомендации относительно реализации разбиений. Описано будет лишь одно из возможных решений (другое основано на присоединении к каждому узлу фантомного четвертого потомка).

В ходе перемещения вниз по дереву алгоритм вставки не обращает внимания на то, заполнены встреченные им узлы или нет. Он просто идет вниз по дереву, пока не найдет подходящий листовой узел. Если лист не полон, то алгоритм вставки вставляет новое значение. Если же лист полон, необходимо внести изменения в структуру дерева для освобождения места. Для этого алгоритм вызывает метод `split()`. В аргументах метода могут передаваться полный листовой узел и новый элемент. Метод `split()` выполняет разбиение и вставляет новый элемент в новый листовой узел.

Если `split()` обнаруживает, что родитель листового узла полон, он рекурсивно вызывает себя для разбиения родителя. Рекурсивные вызовы продолжаются до тех пор, пока не будет найден не-листовой или корневой узел. Возвращаемое значение `split()` — новый правый узел — может использоваться предыдущими экземплярами `split()`.

Программирование разбиения усложняется несколькими факторами. В дереве 2-3-4 три распределяемых элемента изначально отсортированы, а в дереве 2-3 ключ нового элемента нужно сравнить с ключами двух элементов листового узла; распределение трех элементов зависит от результатов сравнения.

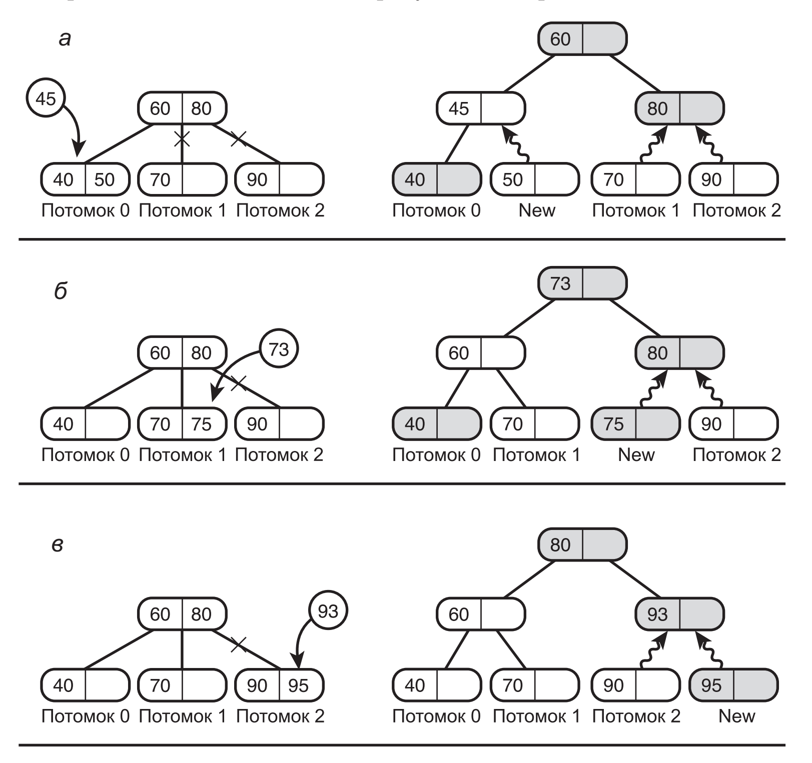


Рис. 10.21. Связывание потомков: *а* — разбиение потомка 0; *б* — разбиение потомка 1; *в* — разбиение потомка 2

Кроме того, в результате разбиения родителя создается второй родитель, так что теперь у нас имеется левый (исходный) родитель и вновь созданный правый родитель. Необходимо превратить связи одного родителя с тремя потомками в двух родителей с двумя потомками каждый. Возможны три случая в зависимости от того, какой потомок (0, 1 или 2) подвергается разбиению. Ситуация представлена на рис. 10.21.

На рисунке новые узлы, созданные в результате разбиения, окрашены в серый цвет, а новые связи обозначены извилистыми линиями.

Внешнее хранение

Деревья 2-3-4 относятся к категории многопутевых деревьев, имеющих более двух потомков и более одного элемента данных. Другой представитель многопутевых деревьев — B-дерево — удобен при нахождении данных во *внешней памяти*, то есть в некоторой разновидности дисковой системы (скажем, на жестком диске).

Раздел открывается описанием различных аспектов внешнего хранения файлов. Мы рассмотрим простейший метод организации внешних данных: *последовательное хранение*. Далее будут рассмотрены B-деревья; мы разберемся, почему они так хорошо подходят для работы с дисковыми файлами. Глава завершается описанием другого подхода к внешнему хранению данных — *индексирования,* — которое может использоваться как само по себе, так и в сочетании с B-деревьями.

Также будут затронуты другие аспекты внешнего хранения данных — такие, как методы поиска. Следующая глава посвящена другому механизму организации внешнего хранения — *хешированию*. Подробности реализации внешнего хранения зависят от операционной системы, языка программирования и даже оборудования, использованного в конкретной установке, поэтому материал этого раздела имеет более общий характер, чем большая часть материала книги.

Обращение к внешним данным

Все структуры данных, упоминавшиеся ранее, исходили из предположения, что данные полностью хранятся в оперативной памяти компьютера. Однако во многих ситуациях объем обрабатываемых данных слишком велик, чтобы эти данные могли одновременно уместиться в оперативной памяти. В таких случаях требуется другая разновидность хранения данных. Как правило, в дисковых файлах может храниться намного больший объем информации, чем в оперативной памяти; это возможно благодаря относительно низкой удельной стоимости хранения.

Конечно, у дисковых файлов есть и другое преимущество: долгосрочный характер хранения. При выключении компьютера (или сбое питания) все содержимое оперативной памяти пропадает. Дисковые файлы могут сколь угодно долго храниться при отсутствии питания. Тем не менее нас прежде всего интересуют различия в потенциальном объеме хранимых данных.

Недостаток внешнего хранения заключается в том, что дисковые устройства работают намного медленнее оперативной памяти. Из-за различий в скорости для эффективной работы с данными приходится применять другие методы.

Рассмотрим пример внешнего хранения данных. Допустим, вы пишете программу для работы с телефонной книгой среднего города — скажем, на 500 000 записей. Каждая запись содержит имя, адрес, номер телефона и другие атрибуты, используемые телефонной компанией. Для хранения одной записи требуется 512 байт. Размер файла базы данных составит 500 000 × 512, то есть 256 000 000 байт, или 256 мегабайт. Будем считать, что они не помещаются в оперативной памяти, но легко могут храниться на диске.

Итак, на жестком диске хранится большой объем информации. Как структурировать эту информацию, чтобы обеспечить обычные желательные характеристики: быстроту поиска, вставки и удаления?

При поиске ответов необходимо учитывать два факта. Во-первых, обращение к данным на диске происходит намного медленнее, чем обращение к данным в основной памяти. Во-вторых, при чтении данных вы будете обращаться сразу к нескольким записям. Рассмотрим эти два момента подробнее.

Медленное обращение

Основная память компьютера работает на электронном уровне. Обращение к любому байту происходит так же быстро, как и обращение к любому другому байту, — за долю микросекунды (миллионной доли секунды).

С дисковыми устройствами дело обстоит сложнее. Данные хранятся на круговых дорожках вращающегося диска, по аналогии с дорожками на компакт-дисках или на старых пластинках.

Чтобы обратиться к некоторому блоку данных на диске, сначала необходимо подвести головку чтения/записи к нужной дорожке. Позиционирование головки осуществляется с помощью шагового привода или другого аналогичного устройства; это механическая операция, выполнение которой занимает несколько миллисекунд (тысячных долей секунды).

После завершения позиционирования головка чтения/записи должна дождаться прохождения нужного участка дорожки. Средняя продолжительность ожидания составляет половину оборота диска. Даже если диск вращается с частотой 10 000 оборотов в минуту, до начала чтения данных проходит еще около трех миллисекунд. Сам процесс чтения (или записи) занимает еще несколько миллисекунд.

Таким образом, среднее время обращения к данным на диске составляет около 10 миллисекунд. Это примерно в 10 000 раз медленнее, чем обращение к оперативной памяти.

С каждым годом технологический прогресс ускоряет обращения к диску, но обращения к оперативной памяти ускоряются еще быстрее, поэтому расхождения между скоростью обращения к данным со временем только растут.

Блочные операции

Если головка чтения/записи находится в правильной позиции, а процесс чтения (или записи) успешно инициирован, устройство может довольно быстро передать в основную память большой объем данных. По этой причине, а также для упрощения механизма управления устройством данные хранятся на диске в блоках. В зависимости от системы также встречаются другие термины, например *страницы* или *кластеры*; мы будем использовать термин «блок».

Дисковое устройство всегда читает или записывает данные объемом не менее одного блока. Конкретный размер блока в байтах зависит от операционной системы, объема диска и других факторов, но обычно он равен степени 2. Будем считать, что в нашем примере с телефонной книгой будет использоваться блок размером 8192 байта (2^{13}). Следовательно, для хранения базы данных потребуется 256 000 000 байт/8192 байта на блок = 31 250 блоков.

Максимальная эффективность чтения/записи в программах достигается при выполнении операций с объемами данных, кратными размеру блока. По запросу на чтение 100 байт система прочитает целый блок (8192 байта) и проигнорирует все, кроме 100 байт. Или если запросить 8200 байт, система прочитает два блока (16 384 байт) и проигнорирует почти половину из них. Структура, при которой программа работает с целыми блоками, оптимизирует ее быстродействие.

Если в примере с телефонной книгой размер одной записи составляет 512 байт, в одном блоке может храниться 16 записей (8192/512), как показано на рис. 10.22. Таким образом, для достижения максимальной эффективности записи должны читаться группами по 16 (или в количестве, кратном этому числу).

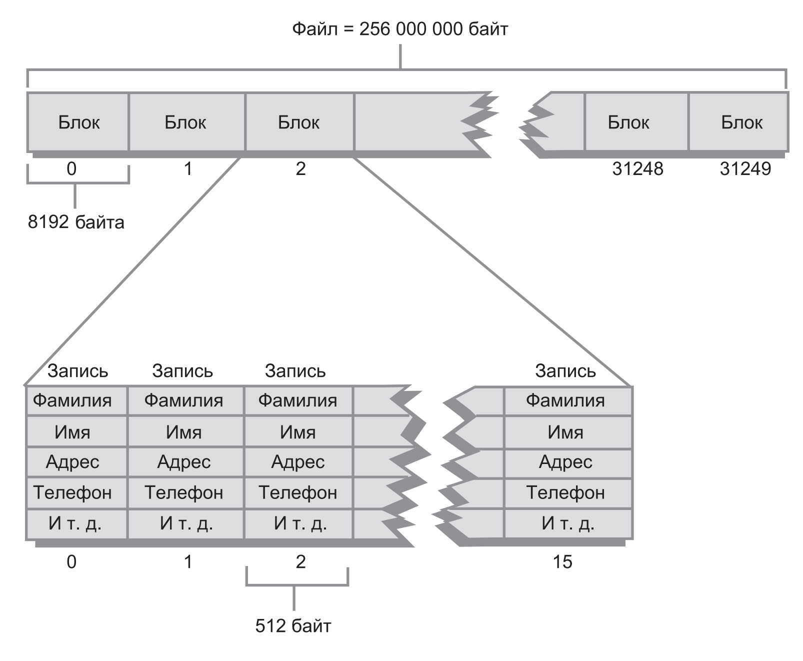


Рис. 10.22. Блоки и записи

Также обратите внимание на то, что размер записи удобно делать кратным двум, чтобы блок всегда вмещал целое количество записей.

Конечно, приведенные здесь размеры записей, блоков и так далее служат исключительно для демонстрационных целей; на практике они сильно зависят от количества и размера записей, а также других программных и аппаратных факторов. Достаточно часто встречаются блоки, содержащие сотни записей, а размер записи может быть больше или меньше 512 байт.

Когда головка чтения/записи находится в нужной позиции, блок считывается относительно быстро — обычно за несколько миллисекунд. Таким образом, время обращения к диску для чтения или записи блока не так уж сильно зависит от размера блока. Соответственно чем больше блок, тем эффективнее выполняется операция чтения или записи блока (если все записи в блоке будут использованы программой).

Последовательное хранение

Самый очевидный способ хранения данных телефонной книги в файле основан на упорядочении всех записей по некоторому ключу (скажем, в алфавитном порядке фамилий). Запись с фамилией Aardvark ставится на первое место и т. д. (рис. 10.23).

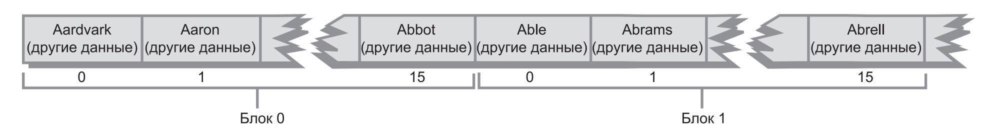


Рис. 10.23. Последовательное хранение

Поиск

Чтобы найти в последовательном файле конкретную фамилию (например, *Smith*), можно воспользоваться алгоритмом двоичного поиска. Сначала читается блок записей из середины файла. Операция читает сразу 16 записей в буфер размером 8192 байта, находящийся в оперативной памяти.

Если ключи прочитанных записей (например, *Keller*) предшествуют искомому ключу в алфавитном порядке, алгоритм переходит к позиции 3/4 файла (*Prince*) и читает блок записей с этой позиции; если ключи следуют после искомого, происходит переход к точке 1/4 (*DeLeon*). Выполняя многократное деление диапазона надвое, алгоритм рано или поздно находит искомую запись.

Как было показано в главе 2, двоичный поиск в оперативной памяти требует $\log_2 N$ сравнений, что для 500 000 элементов составит около 19 сравнений. Если каждое сравнение выполняется, допустим, за 10 микросекунд, вся процедура займет 190 микросекунд, или около 2/10 000 секунды — можно сказать, почти мгновенно.

Однако данные, с которыми мы имеем дело, хранятся на диске. Так как каждое обращение к диску занимает относительно много времени, важнее сосредоточиться на количестве обращений к диску, а не на количестве отдельных записей. Время

чтения блока записей значительно превышает время поиска среди 16 записей, хранящихся в памяти.

Обращения к диску занимают больше времени, чем обращения к памяти, но с другой стороны, с диска читается сразу целый блок, а количество блоков намного меньше количества записей. В нашем примере база данных состоит из 31 250 блоков. Логарифм этого числа по основанию 2 равен приблизительно 15; следовательно, теоретически для нахождения нужной записи потребуется около 15 обращений к диску.

Реальное значение оказывается несколько меньше теоретического. На начальных стадиях двоичного поиска одновременное нахождение нескольких записей в памяти не влияет на скорость, потому что следующее обращение относится к другой позиции файла, находящейся достаточно далеко от текущей. Но при приближении к нужной записи может оказаться, что следующая запись уже находится в памяти, потому что она хранится в уже прочитанном блоке из 16 записей. Это обстоятельство сокращает количество необходимых обращений на два или около того. Таким образом, для поиска записи хватает 13 обращений к диску (15-2), что для 10 миллисекунд на обращение составляет 130 миллисекунд, или 1/7 секунды. Это намного больше, чем при прямых обращениях к памяти, но все еще не так плохо.

Вставка

К сожалению, со вставкой (или удалением) элементов в последовательно упорядоченных файлах дело обстоит намного хуже. Из-за упорядоченности данных каждая из операций требует перемещения в среднем половины записей, а следовательно, около половины блоков.

Перемещение каждого блока требует двух обращений к диску: для чтения и для записи. После обнаружения позиции вставки содержащий ее блок читается в буфер, находящийся в памяти. Последняя запись блока сохраняется во временной переменой, а записи, находящиеся справа от позиции вставки, сдвигаются на одну позицию, чтобы освободить место для вставляемой записи. Затем содержимое буфера снова записывается в файл на диске.

После этого в буфер читается второй блок. Его последняя запись сохраняется во временной переменной, остальные записи сдвигаются вверх, а последняя запись из предыдущего блока вставляется в начало буфера. Далее содержимое буфера снова записывается на диск. Процесс продолжается до тех пор, пока на диск не будут перезаписаны все блоки за позицией вставки.

Если база данных состоит из 31 250 блоков, как в нашем примере, вставка/удаление потребует (в среднем) чтения и записи 15 625 из них. Если каждая операция чтения/записи занимает около 10 миллисекунд, то вставка одной записи займет более 5 минут. При вставке в базу данных тысяч новых записей такие затраты времени неприемлемы.

У последовательного хранения есть еще один недостаток: оно быстро работает только для одного ключа. В нашем примере файл упорядочен по фамилиям; а если потребуется найти в нем конкретный телефонный номер? Воспользоваться двоичным поиском уже не удастся, потому что данные упорядочены по имени.

Придется последовательно перебирать все содержимое файла, блок за блоком. Поиск потребует чтения в среднем половины блоков, что составит около 2,5 минуты — слишком низкое быстродействие для простого поиска. Необходим более эффективный способ хранения данных на диске.

B-деревья

Какая структура записей в файле обеспечит хорошее время поиска, вставки и удаления? Мы уже видели, что деревья хорошо подходят для структурирования данных в памяти. Нельзя ли применить их для хранения данных в файлах?

Оказывается, можно, но для хранения внешних данных потребуется новая разновидность деревьев. Такие деревья относятся к категории многопутевых и отчасти напоминают деревья 2-3-4, но их узлы могут содержать намного большее количество элементов данных; эти деревья называются *B-деревьями*. Идею об использовании B-деревьев для внешнего хранения впервые выдвинули Р. Байер и Э. М. Маккрейт в 1972 году. (Строго говоря, деревья 2-3 и деревья 2-3-4 являются B-деревьями порядков 3 и 4 соответственно, но термином «B-дерево» обычно обозначаются деревья с много большим количеством потомков на узел.)

Один блок на узел

Зачем нам нужно столько элементов на узел? Мы уже видели, что обращения к диску оказываются наиболее эффективными при чтении и записи целых блоков. В дереве данные хранятся в узлах, поэтому логично хранить в каждом узле дерева целый блок данных. В этом случае чтение узла позволит получить доступ к максимальному объему данных за минимальное время.

Сколько данных можно хранить в узле? При простом хранении 512-байтовых записей данных в телефонной книге в одном 8192-байтовом блоке помещалось 16 записей. Однако в дереве также понадобится хранить ссылки на другие узлы (то есть ссылки на другие блоки, потому что узел соответствует блоку). В деревьях, хранимых в памяти (как в тех, которые рассматривались в предыдущих главах), такие ссылки являются аналогами указателей в языках типа C++ и ссылаются на узлы, находящиеся в других областях памяти. Для дерева, хранимого в дисковом файле, ссылки представляют собой номера блоков в файле (от 0 до 31 249 в примере с телефонной книгой). Номера блоков можно хранить в поле типа `int` — 4-байтового типа, который позволяет адресовать до двух миллиардов блоков, чего достаточно для большинства файлов.

Теперь разместить 16 512-байтовых записей в блоке уже не удастся, потому что понадобится дополнительное место для хранения ссылок на потомков. Количество записей можно было бы сократить до 15, но, как говорилось ранее, наиболее эффективно четное количество записей на блок, поэтому (после соответствующего обсуждения с руководством) размер записи сокращается до 507 байт. Количество ссылок на потомков будет равно 17 (на единицу больше количества элементов данных), так что для хранения ссылок потребуется 68 байт (17 × 4). Таким образом, в блоке размещаются 16 507-байтовых записей и еще 12 байт остаются свободными

(507 × 6 + 68 = 8180). Блок такого дерева и соответствующее представление узла изображены на рис. 10.24.

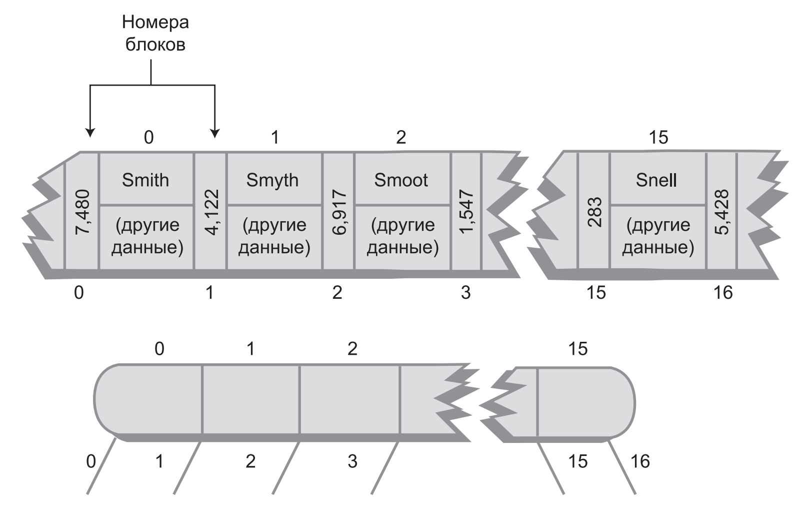


Рис. 10.24. Узел B-дерева порядка 17

Внутри каждого узла данные упорядочиваются последовательно по ключам, как и в деревьях 2-3-4. Структура B-дерева похожа на структуру дерева 2-3-4, не считая большего количества элементов на узел и большего количества ссылок на потомков. Порядком B-дерева называется количество возможных ссылок на потомков в каждом узле. В нашем примере оно равно 17, а дерево соответственно является B-деревом 17-го порядка.

Поиск

Поиск записи с заданным ключом происходит практически так же, как в деревьях 2-3-4, находящихся в памяти. Сначала блок, содержащий корневой узел, читается в память. Затем алгоритм поиска проверяет каждую из 15 записей (или если узел не полон — все фактически наличествующие записи), начиная с 0. При обнаружении записи с большим ключом алгоритм переходит к потомку, ссылка на которого находится между текущей и предыдущей записью.

Процесс продолжается до тех пор, пока не будет найден правильный узел. При достижении листового узла без обнаружения заданного ключа считается, что попытка поиска завершилась неудачей.

Вставка

Процесс вставки в B-дереве больше напоминает вставку в дереве 2-3, нежели в дереве 2-3-4. Вспомните, что в дереве 2-3-4 многие узлы не полны и фактически содержат только один элемент данных. В частности, разбиение всегда создает два узла, каждый из которых содержит только один элемент. Для B-деревьев такой способ не оптимален.

В B-деревьях узлы должны быть по возможности полными, чтобы при каждом обращении к диску, при котором читается весь узел, возвращался максимальный объем данных. Для достижения этой цели процесс вставки отличается от аналогичного процесса деревьев 2-3-4 в трех отношениях:

- ✦ Разбиение узла делит элементы данных пополам: одна половина переходит во вновь созданный узел, а вторая остается в старом узле.
- ✦ Разбиение узлов производится не сверху вниз, а снизу вверх, как в дереве 2-3.
- ✦ Как и в дереве 2-3, наверх переходит не средний элемент узла, а средний элемент последовательности, образованной элементами узла и новым элементом.

Для демонстрации этих особенностей процесса вставки мы построим небольшое B-дерево (рис. 10.25). Места для реалистичного количества записей на узел недостаточно, поэтому мы ограничимся четырьмя записями; таким образом, строится B-дерево порядка 5.

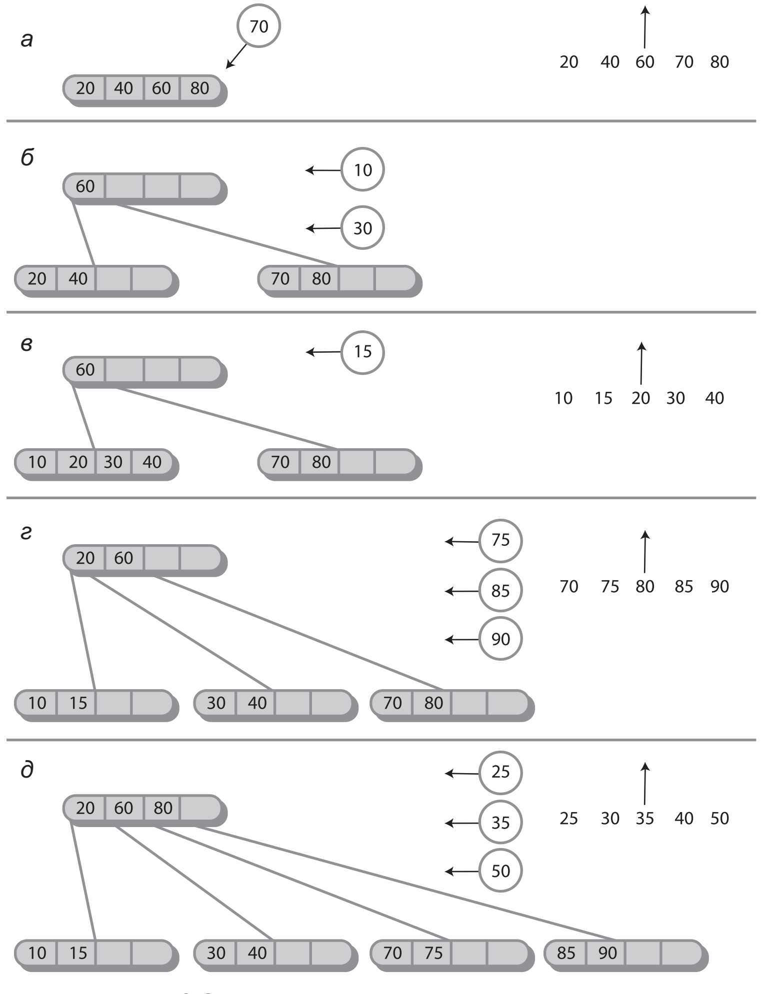


Рис. 10.25. Построение B-дерева (см. также с. 474)

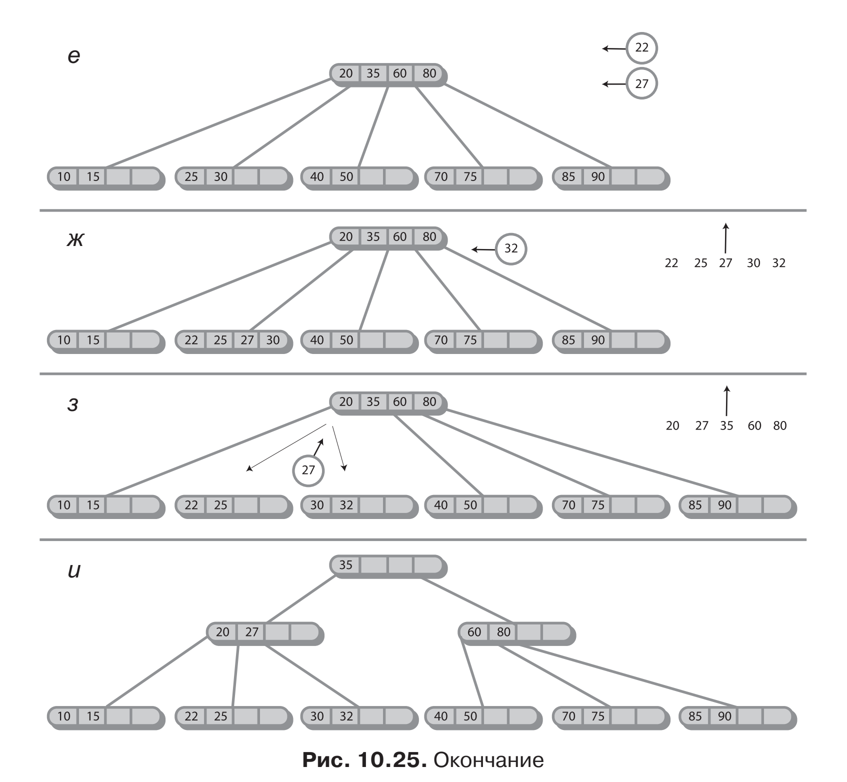


Рис. 10.25. Окончание

На рис. 10.25, *а* корневой узел уже полон; элементы с ключами 20, 40, 60 и 80 уже вставлены в дерево. Вставка нового элемента данных с ключом 70 приводит к разбиению узла. Так как разбивается корневой узел, создаются два новых узла (как в дереве 2-3-4): новый корневой узел и новый узел справа от разбиваемого.

Чтобы решить, куда должны перейти элементы данных, алгоритм вставки упорядочивает пять ключей во внутреннем буфере. Четыре ключа взяты из разбиваемого узла, а пятый принадлежит вставляемому элементу. На рис. 10.25 эти последовательности из пяти элементов выводятся справа от дерева (20, 40, 60, 70, 80 для первого шага).

Средний элемент последовательности (60 на первом шаге) поднимается в новый корневой узел. (На рисунке стрелка показывает, что средний элемент перемещается наверх.) Все элементы слева от среднего остаются в разбиваемом узле, а все элементы справа переходят в новый узел справа. Результат показан на рис. 10.25, *б*. (В нашем примере с телефонной книгой в каждый узел-потомок переходят восемь элементов вместо двух, как на рисунке.)

На рис. 10.25, *б* вставляются еще два элемента 10 и 30. Они заполняют левого потомка, как показано на рис. 10.25, *в*. Вставка следующего элемента 15 приводит к разбиению левого потомка; результат показан на рис. 10.25, *г*. Элемент 20 переходит вверх в корневой узел.

Затем в дерево вставляются еще три элемента — 75, 85 и 90. Первые два заполняют третьего потомка, а третий разбивает его, что приводит к созданию нового узла и повышению среднего элемента 80 до корня. Результат показан на рис. 10.25, *д*.

Наконец, в дерево добавляются еще три элемента — 25, 35 и 50. Первые два заполняют второго потомка, а третий разбивает его, что приводит к созданию нового узла и повышению среднего элемента 35 до корня (рис. 10.25, *е*).

Корневой элемент заполнен. Впрочем, последующие вставки могут и не привести к разбиению узла, потому что разбиение происходит только при вставке нового узла в полный узел, а не при обнаружении полного узла при перемещении вниз по дереве в процессе поиска. Соответственно элементы 22 и 27 вставляются во второго потомка без каких-либо разбиений, как показано на рис. 10.25, *ж*.

Однако вставка следующего элемента 32 приводит к разбиению (даже двум разбиениям). Второй потомок полон, поэтому он разбивается (рис. 10.25, *з*). Элементу 27, повышаемому в результате разбиения, некуда перемещаться; приходится выполнить разбиение корневого узла, что приводит к конфигурации на рис. 10.25, *и*.

Обратите внимание: в процессе вставки ни один узел (кроме корневого) не заполняется менее чем на половину. Как упоминалось ранее, это повышает эффективность работы с данными, потому что операция чтения узла всегда возвращает значительный объем данных.

Эффективность B-деревьев

При таком количестве записей на узел и узлов на уровне, а также особенностей хранения данных на диске операции с B-деревьями выполняются очень быстро. В нашем примере с телефонной книгой база данных содержат 500 000 записей. Все узлы B-дерева заполнены минимум наполовину, поэтому они содержат не менее 8 записей и 9 ссылок на потомков. Высота дерева получается немногим меньше $\log_9 N$ (логарифм N по основанию 9), где $N = 500\ 000$. Результат равен 5972, а значит, дерево будет содержать около 6 уровней.

Итак, при использовании B-дерева любая запись в файле из 500 000 записей может быть найдена за 6 обращений к диску. Если продолжительность каждого обращения составляет около 10 миллисекунд, получается 60 миллисекунд, или 6/100 секунды. Получается намного быстрее, чем при двоичном поиске в последовательно упорядоченном файле.

Чем больше записей хранится в узле, тем меньше уровней в дереве. Мы уже видели, что B-дерево состоит из 6 уровней, хотя каждый узел содержит только 16 записей. С другой стороны, двоичное дерево с теми же 500 000 элементами состояло бы из 19 уровней, а дерево 2-3-4 — из 10 уровней. Если бы блок мог содержать сотни записей, это привело бы к дополнительному сокращению количества уровней и дальнейшему улучшению времени доступа.

Хотя поиск по B-дереву происходит быстрее, чем в последовательных файлах, преимущества B-деревьев в полной мере проявляются при вставке и удалении.

Начнем с вставки в B-дереве, не требующей разбиения узлов. Ввиду большого количества записей на узел этот сценарий является наиболее вероятным. В нашем примере с телефонной книгой для определения позиции вставки требуется всего 6 обращений к диску. Затем еще одно обращение записывает блок с добавленной записью обратно на диск — итого 7 обращений.

Теперь посмотрим, что происходит при разбиении узла. Разбиваемый узел необходимо прочитать, извлечь из него половину записей и записать обратно на диск.

Созданный узел записывается на диск; родителя необходимо прочитать и после вставки записи, перемещенной вверх, записать обратно. К шести обращениям, необходимым для поиска позиции вставки, добавляется еще 5 — итого 12. По сравнению с 500 000 обращений, необходимых для вставки в последовательном файле, выигрыш получается весьма значительным.

В некоторых разновидностях B-деревьев записи хранятся только в листовых узлах. Не-листовые узлы содержат только ключи и номера блоков. Такая структура может ускорить операции с деревом, потому что в каждом блоке хранится намного больше номеров блоков. Такое дерево более высокого порядка содержит меньше уровней, а доступ к нему ускоряется. С другой стороны, существование двух видов узлов, листовых и не-листовых, усложняет программирование.

Индексирование

Другой способ ускорения обращений к файлам основан на последовательном хранении записей в сочетании с файловым индексом. Индекс представляет собой список пар «ключ/блок», упорядоченный по значениям ключа. Напомним, что в нашем исходном примере 500 000 записей (по 512 байт каждые 16 записей) хранились в одном блоке — итого 31 250 блоков. Если предположить, что ключом поиска является фамилия, то каждый элемент индекса состоит из двух частей:

- Ключ (например, *Jones*).
- Номер блока, в котором хранится запись для фамилии *Jones*. Номера блоков лежат в диапазоне от 0 до 31 249.

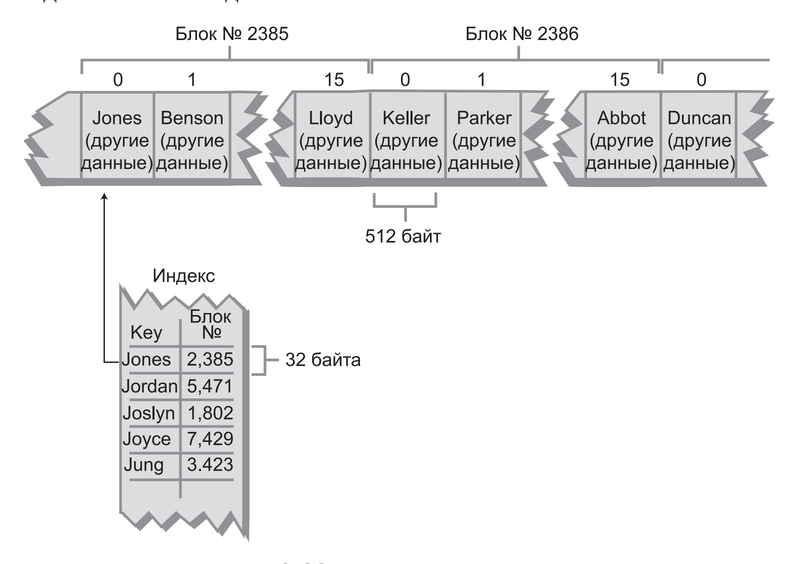


Рис. 10.26. Индексированный файл

Допустим, в качестве ключа используется строка длиной 28 байт (достаточно для большинства фамилий), а для номера блока выделяется 4 байта (тип `int` в Java).

Таким образом, каждый элемент индекса занимает 32 байта — 1/16 от объема каждой записи данных.

Элементы индекса упорядочиваются последовательно по фамилиям. Исходные записи на диске могут располагаться в любом удобном порядке. Обычно это означает, что новые записи просто присоединяются в конец файла, а данные упорядочиваются по времени вставки. Структура индексированной базы данных изображена на рис. 10.26.

Хранение индекса в памяти

Индекс занимает намного меньше места, чем файл с данными. Он даже может оказаться достаточно малым, чтобы полностью поместиться в оперативной памяти. В нашем примере база данных содержит 500 000 записей. Каждая запись представлена 32-байтовым элементом индекса, а значит, индекс будет занимать 32 × 500 000, или 1 600 000 байт (1,6 Мбайт). В современных компьютерах такой объем данных легко помещается в памяти. При этом индекс может храниться на диске, но загружаться в память при запуске программы. В дальнейшем все операции с индексом выполняются в памяти. В конце каждого рабочего дня (а может, и чаще) индекс записывается обратно на диск для долгосрочного хранения.

Поиск

При хранении индекса в памяти операции с файлом телефонной книги выполняются намного быстрее, чем при последовательном хранении записей в файле. Например, двоичный поиск требует 19 обращений к индексу. Если каждое обращение занимает 20 микросекунд, итоговое время составит около 4/10 000 секунды. Также приходится учитывать неизбежные затраты времени на чтение записи из файла после того, как номер блока был найден в индексе, но речь идет всего об одном обращении к диску за время около 10 миллисекунд.

Вставка

Вставка нового элемента в индексированный файл выполняется за два шага. Сначала в основной файл вставляется запись данных, а затем в индекс вставляется элемент индекса, состоящий из ключа и номера блока, в котором хранится новая запись.

Вследствие последовательной упорядоченности индекса вставка нового элемента требует перемещения в среднем половины элементов. Если считать, что перемещение одного байта в памяти занимает две микросекунды, вставка нового элемента занимает 250 000 × 32 × 2, или около 16 секунд. Сравните с пятью минутами для неиндексированного последовательного файла. (Обратите внимание: в основном файле записи не перемещаются, новая запись просто добавляется в конец файла.)

Конечно, для хранения индекса в памяти можно воспользоваться и более сложной структурой данных, например двоичным деревом, деревом 2-3-4 или красно-черным деревом. Такие решения значительно сокращают время вставки и удаления.

В любом случае решения с хранением индекса в памяти работают намного быстрее, чем решения с последовательным файлом. Иногда они даже превосходят по скорости B-деревья.

При вставке в индексированный файл все операции с диском относятся только к новой записи. Обычно последний блок файла читается в память, к нему присоединяется новая запись, и блок записывается на прежнее место. В процессе используется всего два обращения к файлу.

Параллельные индексы

К преимуществам индексирования относится и возможность построения для одного файла нескольких индексов с разными ключами. В одном индексе в качестве ключа используется фамилия, в другом — телефонный номер, в третьем — адрес, и т. д. Поскольку индекс занимает относительно малый объем по сравнению с исходным файлом, построение дополнительных индексов не приводит к значительному увеличению затрат памяти. С другой стороны, оно создает дополнительные проблемы при удалении данных из файла, потому что соответствующие элементы должны быть удалены из всех индексов, но мы не будем рассматривать эту тему.

Индекс не помещается в памяти

Если индекс слишком велик для размещения в памяти, он делится на блоки и сохраняется на диске. Для больших файлов сам индекс может храниться в форме B-дерева. В основном файле записи хранятся в любом удобном порядке.

Такая структура бывает очень эффективной. Присоединение записей в конец основного файла выполняется очень быстро; вставка элемента индекса в новую запись тоже происходит быстро, так как индекс хранится в форме дерева. Результатом является очень быстрый поиск и вставка в больших файлах.

Если индекс структурирован в форме B-дерева, то каждый узел содержит n ссылок на потомков и $n - 1$ элемент данных. Ссылки на потомков представляют собой номера блоков других узлов в индексе. Элементы данных состоят из ключа и ссылки на блок в основном файле. Не путайте эти две разновидности ссылок на блоки.

Сложные критерии поиска

При сложных критериях поиска реально возможно только одно решение: последовательное чтение каждого блока в файле. Допустим, в нашем примере с телефонной книгой потребовалось получить список всех записей с именем *Frank* и местом жительства *Springfield*, у которых телефонный номер содержит три цифры 7.

Файл, упорядоченный по фамилиям, в этом случае полностью бесполезен. Даже если бы в нашем распоряжении были индексные файлы, упорядоченные по именам и городам, все равно не было бы удобного способа найти записи, содержащие одновременно *Frank* и *Springfield*. В таких ситуациях (вполне стандартных для баз данных) задача обычно быстрее всего решается чтением файла блок за блоком и проверкой каждой записи на соответствие критериям.

Сортировка внешних файлов

Для сортировки внешних данных предпочтителен алгоритм сортировки слиянием. Это объясняется тем, что обращения к диску, как и обращения к памяти в других алгоритмах сортировки, чаще происходят в смежных записях, а не в случайных частях файла.

Как говорилось в главе 6, «Рекурсия», метод сортировки слиянием работает рекурсивно, вызывая себя для сортировки уменьшающихся последовательностей. После того как будут отсортированы две наименьших последовательности (по одному байту каждая в версии для внутренней памяти), они будут объединены в отсортированную последовательность двойной длины. Далее происходит последовательное объединение последовательностей все большей длины, пока не будет отсортирован весь файл.

Вариант с внешним хранением реализуется аналогично, однако наименьшей последовательностью, которая читается с диска, является блок записей, поэтому процесс состоит из двух фаз. В первой фазе читается блок, его записи проходят внутреннюю сортировку, а полученный отсортированный блок записывается обратно на диск. Процесс продолжается до тех пор, пока не будут отсортированы все блоки.

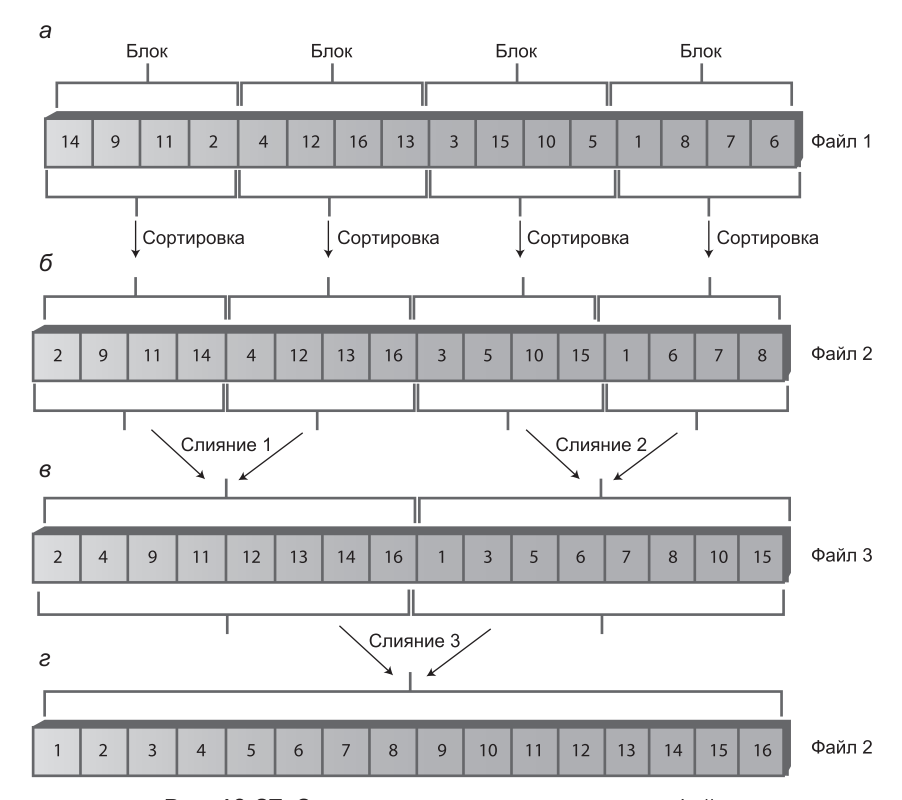


Рис. 10.27. Сортировка слиянием для внешнего файла

Во второй фазе два отсортированных блока читаются с диска, объединяются в последовательность и записываются обратно на диск. Процесс продолжается до тех пор, пока не будут объединены все пары блоков. Затем каждая пара 2-блоковых последовательностей объединяется в 4-блоковую последовательность. Размер отсортированных последовательностей каждый раз увеличивается вдвое, пока не будет отсортирован весь файл.

На рис. 10.27 изображен процесс сортировки слиянием для внешнего файла. Файл состоит из 4 блоков по 4 записи, итого 16 записей. Будем считать, что во внутренней памяти могут поместиться только три блока. (Конечно, в реальной ситуации все эти значения будут намного большими.) На рис. 10.27 показано состояние файла перед сортировкой; в каждой записи приводится значение ее ключа.

Внутренняя сортировка блоков

В первом файле все блоки файла проходят внутреннюю сортировку. Для этого блок читается в память и сортируется каким-нибудь алгоритмом внутренней сортировки — например, алгоритм быстрой сортировки (или при меньшем количестве записей — алгоритмом сортировки Шелла или вставки). Результат внутренней сортировки блоков изображен на рис. 10.27, *б*.

Для хранения отсортированных блоков может использоваться другой файл. Будем считать, что объем внешней памяти неограничен; как правило, содержимое исходного файла лучше оставить неизменным.

Слияние

Во второй фазе выполняется слияние отсортированных блоков. На первом проходе каждая пара блоков объединяется в отсортированную 2-блоковую последовательность. Таким образом, блоки 2-9-11-14 и 4-12-13-16 объединяются в блок 2-4-9-11-12-13-14-16, а блоки 3-5-10-15 и 1-6-7-8 объединяются в 1-3-5-6-7-8-10-15. Результат показан на рис. 10.27, *в*. Для хранения результата этого шага сортировки потребуется третий файл.

На втором проходе две последовательности из 8 записей объединяются в последовательность из 16 записей, которая затем записывается обратно в файл 2 (рис. 10.27, *г*). На этом сортировка завершается. Конечно, сортировка файлов большего размера потребует большего количества шагов, пропорционального $\log_2 N$. В ходе слияния можно попеременно использовать два вспомогательных файла (файлы 2 и 3 на рис. 10.22).

Внутренние массивы

Так как во внутренней памяти компьютера могут одновременно находиться только три блока, процесс слияния должен выполняться за несколько этапов. Допустим, в программе используются три массива `arr1`, `arr2` и `arr3`, в каждом из которых может храниться один блок.

При первом слиянии блок 2-9-11-14 читается в `arr1`, а блок 4-12-13-16 читается в `arr2`. Эти два массива объединяются в `arr3` посредством сортировки слиянием.

Но поскольку в arr3 помещается только один блок, массив будет заполнен еще до завершения сортировки. Содержимое заполненного массива записывается на диск, после чего сортировка продолжается, а массив arr3 заполняется снова. Когда сортировка будет завершена, arr3 снова записывается на диск. Ниже подробно расписан процесс выполнения каждой из трех сортировок слиянием.

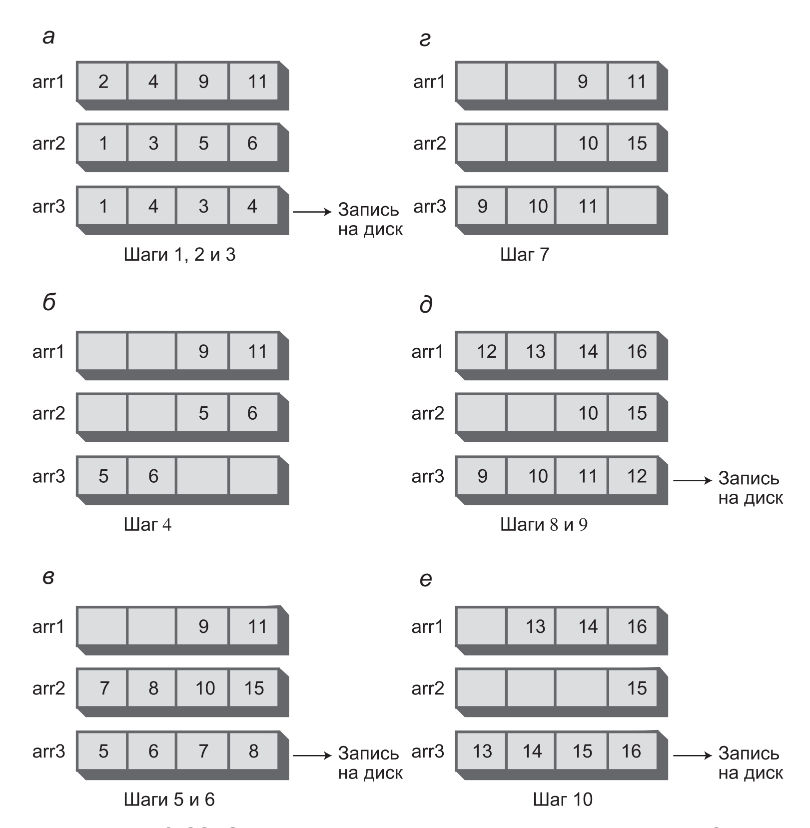


Рис. 10.28. Содержимое массивов в процессе сортировки 3

Сортировка 1:

1. Чтение 2-9-11-14 в arr1.
2. Чтение 4-12-13-16 в arr2.
3. Слияние 2, 4, 9, 11 в arr3; запись на диск.
4. Слияние 12, 13, 14, 16 в arr3; запись на диск.

Сортировка 2:

1. Чтение 3-5-10-15 в arr1.
2. Чтение 1-6-7-8 в arr2.
3. Слияние 1, 3, 5, 6 в arr3; запись на диск.
4. Слияние 7, 8, 10, 15 в arr3; запись на диск.

Сортировка 3:

1. Чтение 2-4-9-11 в arr1.
2. Чтение 1-3-5-6 в arr2.
3. Слияние 1, 2, 3, 4 в arr3; запись на диск.
4. Слияние 5, 6 в arr3 (массив arr2 пуст).
5. Чтение 7-8-10-15 в arr2.
6. Слияние 7, 8 в arr3; запись на диск.
7. Слияние 9, 10, 11 в arr3 (массив arr1 пуст).
8. Чтение 12-13-14-16 в arr1.
9. Слияние 12 в arr3; запись на диск.
10. Слияние 13, 14, 15, 16 в arr3; запись на диск.

Так как последняя серия из 10 шагов получается довольно длинной, будет полезно привести содержимое массивов на момент окончания каждого шага. На рис. 10.28 показано, какие данные хранятся в массивах на разных стадиях сортировки 3.

Итоги

- Узлы многопутевого дерева содержат больше ключей и потомков, чем узлы двоичного дерева.
- Дерево 2-3-4 является многопутевым деревом, каждый узел которого содержит до трех ключей и имеет до четырех потомков.
- В многопутевом дереве ключи в узлах упорядочиваются по возрастанию.
- В дереве 2-3-4 все операции вставки выполняются в листовых узлах, а все листовые узлы находятся на одном уровне.
- В дереве 2-3-4 возможны три разновидности узлов: 2-узлы имеют один ключ и двух потомков, 3-узлы имеют два ключа и трех потомков, а 4-узлы — три ключа и четырех потомков.
- В дереве 2-3-4 не существует 1-узлов.
- При поиске в дереве 2-3-4 просматриваются ключи узла. Если искомый ключ не найден, то алгоритм переходит к потомку 0, если искомый ключ меньше ключа 0; к потомку 1, если искомый ключ лежит в диапазоне между ключом 0 и ключом 1; к потомку 2, если искомый ключ больше ключа 2.
- При вставке в дереве 2-3-4 выполняется разбиение всех полных узлов при перемещении вниз по дереву, в процессе поиска позиции вставки.
- Разбиение корневого узла приводит к созданию двух новых узлов; при разбиении любого другого узла создается один новый узел.

- Высота дерева 2-3-4 увеличивается только при разбиении корневого узла.
- Деревья 2-3-4 и красно-черные деревья связаны отношениями эквивалентности.
- В процессе преобразования дерева 2-3-4 в красно-черное дерево каждый 2-узел преобразуется в черный узел, каждый 3-узел — в черного родителя с красным потомком, а каждый 4-узел — в черного родителя с двумя красными потомками.
- При преобразовании 3-узла в пару «родитель/потомок» родителем может стать любой из двух узлов.
- Разбиение узла дерева 2-3-4 эквивалентно переключению цветов в красно-черном дереве.
- Поворот в красно-черном дереве эквивалентен изменению двух возможных ориентаций (наклонов) при преобразовании 3-узла.
- Высота дерева 2-3-4 меньше $\log_2 N$.
- Время поиска пропорционально высоте.
- В дереве 2-3-4 многие узлы не заполнены даже наполовину.
- Деревья 2-3 похожи на деревья 2-3-4, но их узлы могут содержать только один или два элемента данных с одним, двумя или тремя потомками.
- Чтобы вставить элемент в дерево 2-3, необходимо найти подходящий лист, а затем выполнять разбиения от листа к корню до тех пор, пока не будет найден неполный узел.
- Внешним хранением данных называется хранение данных за пределами оперативной памяти (обычно на диске).
- Внешняя память имеет больший объем, дешевле стоит и медленнее работает, чем оперативная память.
- Данные, находящиеся во внешней памяти, обычно читаются в оперативную память и записываются по одному блоку.
- Данные во внешней памяти могут храниться последовательно, упорядоченными по значениям ключей. Такой способ упорядочения обеспечивает быстрый поиск при медленной вставке (и удалении).
- B-дерево представляет собой многопутевое дерево, каждый узел которого может иметь десятки и сотни ключей и потомков.
- Количество потомков узла B-дерева всегда на единицу больше количества ключей.
- Для достижения оптимального быстродействия B-дерево обычно структурируется таким образом, что в узле хранится один блок данных.
- Для сложных критериев поиска, в которых задействовано несколько ключей, самым практичным решением может оказаться последовательный перебор всех записей в файле.

Вопросы

Следующие вопросы помогут читателю проверить качество усвоения материала. Ответы на них приведены в приложении В.

1. Дерево 2-3-4 называется так потому, что узел может иметь:
   a) трех потомков и четыре элемента данных;
   b) двух, трех или четырех потомков;
   c) двух родителей, трех потомков и четыре элемента;
   d) двух родителей, три элемента и четырех потомков.
2. Преимущество дерева 2-3-4 перед деревом двоичного поиска заключается в том, что оно _________ .
3. Допустим, имеется родительский узел с элементами данных 25, 50 и 75. Если один из потомков содержит элементы со значениями 60 и 70, то это будет потомок с номером ___________ .
4. Элементы данных хранятся исключительно в листовых узлах (Да/Нет).
5. Какое из следующих утверждений *ложно*? При разбиении узла всегда:
   a) создается ровно один новый узел;
   b) в дерево добавляется ровно один новый элемент данных;
   c) один элемент данных перемещается из разбиваемого узла в родительский;
   d) один элемент данных перемещается из разбиваемого узла в создаваемого «брата».
6. Количество уровней в дереве 2-3-4 увеличивается при _________.
7. Что из перечисленного не происходит при поиске в дереве 2-3-4?
   a) разбиение узлов при перемещении вниз в случае необходимости;
   b) выбора потомка для перехода в зависимости от элементов данных узла;
   c) завершение поиска на листовом узле, если искомый ключ не найден;
   d) проверка минимум одного элемента в каждом посещаемом узле.
8. Выполнено разбиение не-корневого узла в дереве 2-3-4. Элемент с каким прежним номером содержится в новом правом потомке — 0, 1 или 2?
9. Разбиение 4-узла в дереве 2-3-4 эквивалентно ________ в красно-черном дереве.
10. Какое из следующих утверждений об операции разбиения узлов в дереве 2-3 (не в дереве 2-3-4!) *ложно*?
    a) если родитель разбиваемого узла полон, то он тоже должен быть подвергнут разбиению;
    b) наименьший элемент разбиваемого узла всегда остается в этом узле;
    c) при разбиении родителя потомок 2 всегда отсоединяется от старого родителя и присоединяется к новому родителю;
    d) процесс разбиения начинается с листового узла и продвигается вверх.

11. С какой сложностью (в O-синтаксисе) выполняются операции с деревом 2-3?
12. При обращении к данным, хранящимся на диске:
    a) вставка выполняется медленно, но позиция для записи данных находится быстро;
    b) перемещение данных с целью освобождения места под новые данные выполняется быстро благодаря возможности одновременного обращения ко многим записям;
    c) удаление данных выполняется особенно быстро;
    d) поиск места для записи данных выполняется относительно медленно, но существует возможность записи большого объема данных.
13. В B-дереве каждый узел содержит ________ элементов данных.
14. Процесс разбиения узлов в B-дереве похож на разбиение в дереве 2-3.
15. В области внешнего хранения данных индексированием называется создание файла, содержащего:
    a) ключи с соответствующими номерами блоков;
    b) записи с соответствующими номерами блоков;
    c) ключи с соответствующими записями;
    d) фамилии с соответствующими ключами.

Упражнения

Упражнения помогут вам глубже усвоить материал этой главы. Программирования они не требуют.

1. Нарисуйте состояние дерева 2-3-4 после выполнения каждой из следующих операций вставки: 10, 20, 30, 40, 50, 60, 70, 80 и 90. Не используйте приложение Tree234 Workshop.
2. Нарисуйте состояние дерева 2-3 после выполнения последовательности вставок из п. 1.
3. Подумайте, как бы вы реализовали удаление узла из дерева 2-3-4.

Программные проекты

В этом разделе читателю предлагаются программные проекты, которые укрепляют понимание материала и демонстрируют практическое применение концепций этой главы. (Как упоминалось во введении, преподаватели могут получить полные решения на сайте издательства.)

10.1. Напишите метод, возвращающий наименьшее значение в дереве 2-3-4.

10.2. Напишите метод, выполняющий симметричный обход дерева 2-3-4. В процессе перехода метод должен выводить упорядоченную последовательность элементов.

10.3. Дерево 2-3-4 может использоваться для сортировки. Напишите метод `sort()`, который получает массив ключей из `main()` и записывает их обратно в порядке сортировки.

10.4. Измените программу `tree234.java` (см. листинг 10.1), чтобы она создавала дерево 2-3 и работала с ним. Программа должна выводить содержимое дерева и поддерживать поиск. Она также должна поддерживать вставку элементов, но только в том случае, если это не требует разбиения родителя листового (разбиваемого) узла. Таким образом, метод `split()` не обязан быть рекурсивным. Программируя метод `insert()`, помните, что до обнаружения узла никакие разбиения не выполняются. Далее, если узел полон, то он подвергается разбиению. Также следует предусмотреть возможность разбиения корневого узла, но только в том случае, если он является листовым. Ограниченная реализация позволит вставить в дерево до девяти элементов, после чего дальнейшая вставка станет невозможной.

10.5. Доработайте проект из п. 10.4. Сделайте метод `split()` рекурсивным, чтобы он нормально обрабатывал ситуации «полный родитель/полный потомок». Такая реализация позволит вставить в дерево неограниченное количество элементов. Учтите, что в усовершенствованной версии `split()` необходимо выполнить разбиение родительского узла, прежде чем метод сможет решить, куда перемещаются элементы и где будут присоединяться потомки.

Глава 11

Хеш-таблицы

Хеш-таблицей называется структура данных, обеспечивающая очень быструю вставку и поиск. На первый взгляд звучит слишком хорошо, чтобы быть правдой: независимо от количества элементов данных вставка и поиск (а иногда и удаление) выполняются за время, близкое к постоянному: $O(1)$ в О-синтаксисе. На практике это лишь несколько машинных команд.

Для пользователя хеш-таблицы обращение к данным происходит практически мгновенно. Все делается настолько быстро, что компьютерные программы часто используют хеш-таблицы при необходимости сделать выборку из десятков тысяч элементов менее чем за секунду (как, например, в системах проверки орфографии).

Хеш-таблицы по скорости значительно превосходят деревья, которые, как вы узнали в предыдущих главах, выполняют операции за с относительно малое время $O(\log N)$. Операции с хеш-таблицами не только быстро выполняются, но и относительно просто программируются.

У хеш-таблиц также имеются свои недостатки. Они реализуются на базе массивов, а массивы трудно расширить после создания. У некоторых разновидностей хеш-таблиц быстродействие катастрофически падает при заполнении таблицы, поэтому программист должен довольно точно представлять, сколько элементов данных будет храниться в таблице (или приготовиться к периодическому перемещению данных в другую хеш-таблицу большего размера — процесс занимает довольно много времени).

Кроме того, при работе с хеш-таблицами не существует удобного способа перебора элементов в определенном порядке (скажем, от меньших к большим). Если вам необходима такая возможность, поищите другую структуру данных.

Но если вам не требуется перебирать элементы в определенном порядке, а размер базы данных можно спрогнозировать заранее, хеш-таблицы не имеют себе равных по скорости и удобству.

Хеширование

Этот раздел содержит краткое введение в хеш-таблицы и хеширование, а также важная концепция отображения диапазона значений ключа на диапазон значений индекса. В хеш-таблице такое отображение осуществляется при помощи хеш-функции. Впрочем, для некоторых видов ключей хеширование не требуется; значения ключей могут использоваться напрямую как индексы массива. Сначала мы рассмотрим эту упрощенную ситуацию, а затем перейдем к применению хеширования в ситуациях, когда ключи не имеют удобного распределения.

Табельные номера как ключи

Предположим, вы пишете программу для отдела кадров небольшой компании — допустим, со штатом в 1000 работников. Каждая запись работника занимает 1000 байт. Таким образом, вся база данных занимает всего один мегабайт и легко поместится в памяти компьютера.

Директор отдела кадров требует как можно более быстрого доступа к любой отдельной записи. Всем работникам в штате фирмы присвоены табельные номера от 1 (учредитель) до 1000 (последний нанятый работник). Табельные номера могут использоваться в качестве ключей для обращения к записям; в быстром доступе по другим ключам необходимости нет. Работники увольняются редко, причем даже после увольнения их записи остаются в базе данных (для учета выходных пособий и т. д.). Какую структуру данных следует использовать в такой ситуации?

Индексы как ключи

База данных легко реализуется в виде простого массива. Каждая запись работника занимает одну ячейку, а индекс ячейки используется в качестве табельного номера. Пример такого массива приведен на рис. 11.1.

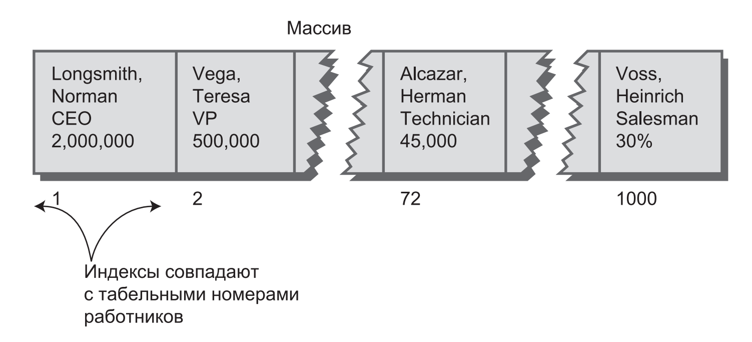


Рис. 11.1. Использование номеров работников в качестве индексов массива

Как известно, обращение к элементу массива по индексу происходит очень быстро. Пользователь, ищущий данные работника *Henry Alcazar*, знает, что этому работнику присвоен табельный номер 72; он вводит это число, и программа немедленно обращается к элементу массива с индексом 72. Все это делается одной командой:

```
empRecord rec = databaseArray[72];
```

Операция добавления нового элемента тоже выполняется очень быстро: элемент просто вставляется за последней занятой ячейкой. Следующая новая запись будет записана в ячейку 1001. И снова вставка новой записи выполняется одной командой:

```
databaseArray[totalEmployees++] = newRecord;
```

Размер массива несколько превышает текущее количество работников для возможного расширения штата, но сколько-нибудь значительного расширения не планируется.

Отсутствие четкой структуры

Благодаря скорости и простоте обращения к элементам в такой базе данных такое решение выглядит весьма привлекательно. Однако в нашей ситуации оно работает только потому, что ключи имеют четко определенную структуру. Записи нумеруются последовательно от единицы до заранее известного максимума, и этот максимум соответствует приемлемому размеру массива. При отсутствии удалений в массиве не появятся пустоты, приводящие к неэффективному расходованию памяти. Новые элементы последовательно добавляются в конец массива, поэтому размер массива не обязан значительно превышать текущее количество элементов.

Словарь

Однако ключи далеко не всегда настолько удобны, как в только что рассмотренном примере. Классическим примером служит словарь. Если вы хотите разместить в памяти компьютера все слова английского языка, от *a* до *zyzzyva* (да, есть и такое слово) с возможностью быстрого обращения, хеш-таблица хорошо подойдет для этой цели.

Хеш-таблицы также часто применяются в компиляторах для работы с таблицами символических имен. В таблице символических имен хранятся все имена переменных и функций, заданные программистом, вместе с адресами их расположения в памяти. Компилятор должен очень быстро обращаться к этим именам, поэтому хеш-таблица в данном случае является предпочтительной структурой данных.

Допустим, вы хотите сохранить в оперативной памяти словарь английского языка на 50 000 слов. Каждое слово должно занимать отдельную ячейку массива, чтобы к каждому слову можно было обратиться по его индексу. Доступ к данным будет очень быстрым, но как связать индексы со словами? Скажем, вы хотите найти в словаре слово *morphosis*; как определить его индекс?

Преобразование слов в числа

Нам нужна система, которая преобразует слово в соответствующий ему индекс. Как известно, компьютеры используют разные схемы числового представления отдельных символов. Одной из таких схем является кодировка ASCII, в которой букве «a» соответствует код 97, букве «b» — код 98 и т. д., до кода 122 для буквы «z».

Однако ASCII-коды занимают диапазон от 0 до 255; среди них имеются представления для символов верхнего и нижнего регистра, знаков препинания и т. д. В английском алфавите всего 26 букв, поэтому мы создадим собственную кодировку — упрощенную, которая может сэкономить память. Допустим, буква «a» представляется кодом 1, буква «b» — кодом 2 и т. д., до «z» (код 26). Пробел представляется 0, итого 27 символов. (В нашем словаре символы верхнего регистра не используются.)

Как объединить коды отдельных букв в число, представляющее целое слово? Существует много разных способов. Мы рассмотрим два типичных решения, их достоинства и недостатки.

Суммирование

Простейший способ преобразования слова в число заключается в обычном суммировании кодов символов. Допустим, мы хотим преобразовать слово *cats* в число. Сначала символы преобразуются в цифры по правилам нашего доморощенного шифра:

```
c = 3
a = 1
t = 20
s = 19
```

Затем полученные цифры суммируются:

$$3 + 1 + 20 + 19 = 43.$$

Итак, в нашем словаре слово *cats* будет храниться в ячейке массива с индексом 43. Другим словам также назначаются индексы, вычисленные аналогичным образом.

Насколько хорошо работает эта схема? В рамках обсуждения ограничимся словами из 10 букв. В этом случае первое слово алфавита «a» будет кодироваться следующей последовательностью (напомним, что пробел кодируется 0):

$$0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 1 = 1.$$

Теоретически последним словом в словаре будет последовательность zzzzzzzzzz (10 букв «z»). Суммируя коды символов, получаем

$$26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 = 260.$$

Таким образом, все коды в словаре лежат в диапазоне от 1 до 260. К сожалению, словарь содержит 50 000 слов, и индексов оказывается явно недостаточно. В каждом элементе массива придется хранить в среднем 192 слова (50 000/260).

Естественно, в рамках схемы «одно слово на один элемент массива» это создаст проблемы. Теоретически в каждом элементе массива можно разместить подмассив или связанный список слов. К сожалению, такое решение приведет к серьезному ухудшению скорости доступа. Обращение к элементу массива будет быстрым, но перебор 192 слов в поисках нужного займет слишком много времени.

Итак, первая попытка преобразования слов в числа оставляет желать лучшего. Слишком много слов преобразуется в одинаковые индексы. (Например, буквы слов *was*, *tin*, *give*, *tend*, *moan*, *tick*, *bails*, *dredge* и сотни других слов в сумме дают 43, как и буквы слова *cats*.) Следовательно, описанный подход не обладает достаточной избирательностью — полученный массив содержит слишком мало элементов. Диапазон возможных индексов необходимо расширить.

Умножение на степени

Попробуем другой способ отображения слов на числа. А если создать массив, в котором каждое слово (даже каждое потенциальное слово от a до zzzzzzzzzz) гарантированно занимает отдельную ячейку?

Для этого необходимо разработать такую схему преобразования, в которой каждый символ слова вносит уникальный вклад в итоговое число.

Вспомните, что в обычном числе, состоящем из нескольких цифр, каждая цифра представляет значение в 10 раз большее, чем цифра справа от нее. Таким образом, запись 7546 в действительности означает

$$7*1000 + 5*100 + 4*10 + 6*1.$$

Или если перейти к записи со степенями 10:

$$7*10^3 + 5*10^2 + 4*10^1 + 6*10^0.$$

(Функции ввода в компьютерных программах выполняют аналогичные серии умножений и сложений для преобразования последовательности цифр, введенной с клавиатуры, в число, хранящееся в памяти.)

В этой схеме число разбивается на цифры, каждая цифра умножается на соответствующую степень 10 (так как существует 10 возможных цифр), после чего произведения суммируются.

Аналогичным образом можно поступить и со словом: разложить его на буквы, преобразовать буквы в числовые эквиваленты, умножить их на соответствующие степени 27 (так как существует 27 возможных символов, включая пробелы) и сложить результаты. В результате для каждого слова генерируется уникальное число.

Допустим, нам потребовалось преобразовать слово *cats* в число. Мы заменяем буквы цифрами так, как было показано ранее, после чего умножаем каждое число на соответствующую степень 27 и суммируем результаты:

$$3*27^3 + 1*27^2 + 20*27^1 + 19*27^0.$$

Вычисление степеней дает

$$3*19\,683 + 1*729 + 20*27 + 19*1,$$

а в результате суммирования кодов букв, умноженных на степени, мы получаем

$$59\,049 + 729 + 540 + 19 = 60\,337.$$

Эта схема действительно генерирует уникальное число для каждого потенциального слова. Мы только что вычислили числовой эквивалент слова из четырех букв. А что произойдет со словами большего размера? К сожалению, диапазон чисел становится очень большим. Наибольшее 10-буквенное слово zzzzzzzzzz преобразуется в

$$26*27^9 + 26*27^8 + 26*27^7 + 26*27^6 + 26*27^5 + 26*27^4 + 26*27^3 + \\ + 26*27^2 + 26*27^1 + 26*27^0.$$

Значение 27^9 само по себе превышает 7 000 000 000 000, так что сумма получается огромной. Массив, хранимый в памяти, не может содержать такого количества элементов.

Недостаток этой схемы заключается в том, что элемент массива выделяется для каждой последовательности букв, независимо от того, существует такое слово в английском языке или нет. Таким образом, в массиве будут зарезервированы

ячейки для «слов» aaaaaaaaaa, aaaaaaaaab, aaaaaaaaac и т. д., вплоть до zzzzzzzzzz. Лишь небольшая часть ячеек будет задействована для хранения «настоящих» слов, а большинство ячеек останется пустым. Ситуация показана на рис. 11.2.

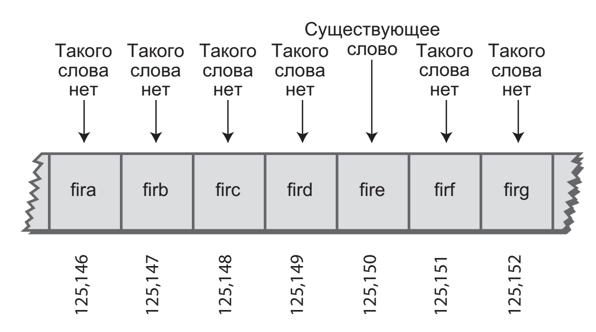


Рис. 11.2. Выделение индекса для каждого потенциального слова

Первая схема (с суммированием кодов букв) генерировала слишком мало индексов. Вторая схема (с суммированием кодов букв, умноженных на степени 27) генерирует слишком много индексов.

Хеширование

Огромный диапазон чисел, полученных по схеме с умножением на степени, необходимо каким-то образом сжать до диапазона, соответствующего приемлемому размеру массива.

Каким должен быть этот размер для массива, в котором хранится словарь английского языка? Если словарь содержит около 50 000 слов, можно предположить, что массив должен содержать приблизительно столько элементов. Однако как выясняется, массив должен содержать примерно вдвое больше ячеек (причины вскоре станут ясны). Итак, нам понадобится массив из 100 000 элементов.

А это означает, что отображение должно сжимать диапазон от 0 до более 7 000 000 000 000 до диапазона от 0 до 100 000. Простейшее решение основано на использовании оператора вычисления остатка от целочисленного деления (`%`).

Следующий пример поможет понять, как работает эта схема. Допустим, числа из диапазона от 0 до 199 (представленные переменной `largeNumber`) должны отображаться в диапазон от 0 до 9 (переменная `smallNumber`). Сокращенный диапазон состоит из 10 чисел (переменная `smallRange`). Верхняя граница «большого» диапазона для нас не существенна (лишь бы числа не превышали максимальное значение переменной). Выражение для выполнения преобразования на языке Java выглядит так:

```
smallNumber = largeNumber % smallRange;
```

Остатки от деления любого числа на 10 всегда лежат в диапазоне от 0 до 9; например, результат выражения 13%10 равен 3, а результат 157%10 равен 7 (рис. 11.3). Диапазон 0–199 преобразуется в диапазон 0–9, то есть происходит сжатие с коэффициентом 20:1.

Аналогичное выражение может использоваться для сжатия очень больших чисел, однозначно представляющих каждое слово английского языка, в индексы массива:

```
arrayIndex = hugeNumber % arraySize;
```

Это выражение является примером *хеш-функции* — функции, преобразующей число из большего диапазона в число из меньшего диапазона. Меньший диапазон соответствует индексам массива. Массив, в который вставляются данные с использованием хеш-функции, называется хеш-таблицей. (О том, как проектируются хеш-функции, более подробно рассказано позднее в этой главе.)

Подведем итог: слово преобразуется в очень большое число, для чего код каждого символа умножается на соответствующую степень 27.

$$\text{hugeNumber} = \text{ch0}*27^9 + \text{ch1}*27^8 + \text{ch2}*27^7 + \text{ch3}*27^6 + \text{ch4}*27^5 + \text{ch5}*27^4 + \text{ch6}*27^3 + \text{ch7}*27^2 + \text{ch8}*27^1 + \text{ch9}*27^0.$$

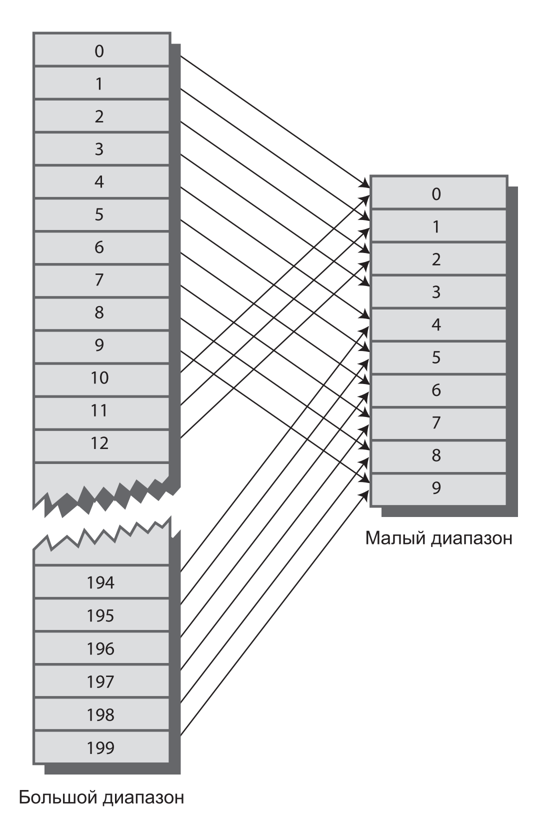


Рис. 11.3. Преобразование диапазона

Затем при помощи оператора вычисления остатка (%) числовой диапазон «сжимается» в диапазон, размер которого вдвое больше количества хранимых элементов. В преобразовании используется хеш-функция:

```
arraySize = numberWords * 2;
arrayIndex = hugeNumber % arraySize;
```

В большом диапазоне каждое число представляет потенциальный элемент данных (последовательность букв), но лишь немногие из этих чисел представляют реально существующие элементы (слова английского языка). Хеш-функция преобразует большие числа в индексы намного меньшего массива. Предполагается, что в нашем массиве на каждые две ячейки будет приходиться одно слово. В некоторых ячейках слов не будет, а в других будет более одного слова.

Практическая реализация этой схемы сталкивается с проблемами — вероятно, переменная `hugeNumber` выйдет за пределы диапазона допустимых значений даже для типа `long`. О том, как решается эта проблема, мы поговорим позднее.

Коллизии

Сокращение диапазона значений имеет свою цену. Схема уже не гарантирует, что после сжатия два слова не будут хешироваться в один индекс массива.

Нечто похожее происходило и при суммировании кодов букв, но тогда все было гораздо хуже. При суммировании было всего 260 возможных результатов (для слов длиной до 10 букв), а теперь диапазон расширяется до 50 000 возможных результатов.

Впрочем, даже в этом случае невозможно избежать хеширования нескольких разных слов в один индекс массива (по крайней мере изредка). Мы надеялись, что на один индекс будет приходиться один элемент данных, но как оказалось, это невозможно. Лучшее, на что можно надеяться, что не слишком много слов будет хешироваться в один индекс.

Предположим, в массив вставляется слово *melioration*. Слово хешируется, вы вычисляете его индекс — и тут выясняется, что ячейка с этим индексом уже занята словом *demystify*, которое имеет точно такой же хеш-код (для заданного размера массива). Такая ситуация, показанная на рис. 11.4, называется *коллизией*.

Может показаться, что из-за возможности коллизий вся схема хеширования теряет смысл, но у проблемы существует несколько обходных решений.

Вспомните, что при определении массива количество зарезервированных ячеек вдвое превышало количество элементов данных. Таким образом, приблизительно половина ячеек остается пустой. Одно из возможных решений при возникновении коллизии заключается в систематизированном поиске пустой ячейки и вставке нового элемента в нее (вместо индекса, полученного в результате применения хеш-функции). Такое решение называется *открытой адресацией*. Если слово *cats* хешируется в индекс 5421, а эта ячейка уже занята словом *parsnip*, можно попытаться вставить *cats* в другую ячейку, например 5422.

Во втором решении (упоминавшемся ранее) создается массив, содержащий связанные списки слов вместо самих слов. При возникновении коллизии новый элемент просто вставляется в список с соответствующим индексом. Этот метод называется *методом цепочек*. Мы рассмотрим открытую адресацию и метод цепочек, а затем вернемся к теме хеш-функций.

До настоящего момента мы ограничивались хешированием строковых данных. В самом деле, многие хеш-таблицы используются для хранения строковых данных.

Однако в хеш-таблицах нередко хранятся и числовые данные, как в нашем примере с табельными номерами. В последующем обсуждении и в приложении Workshop в качестве ключей будут использоваться числа, а не строки. Это сделано для того, чтобы упростить понимание материала и примеры программного кода. Однако следует помнить, что во многих ситуациях эти числа будут строиться на основе строковых данных.

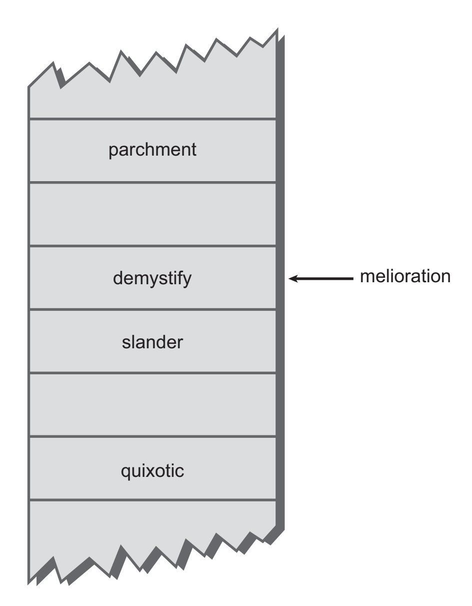


Рис. 11.4. Коллизия

Открытая адресация

Если элемент данных не удается разместить в ячейке с индексом, вычисленным посредством хеш-функции, метод открытой адресации ищет в массиве другую ячейку. Мы рассмотрим три разновидности открытой адресации, различающихся способом поиска следующей свободной ячейки: *линейное пробирование*, *квадратичное пробирование* и *двойное хеширование*.

Линейное пробирование

Алгоритм линейного пробирования последовательно ищет пустые ячейки. Если при попытке вставки элемента выясняется, что ячейка 5421 занята, мы переходим к ячейке 5422, затем к ячейке 5423 и т. д. Индекс последовательно увеличивается до тех пор, пока не будет найдена пустая ячейка. Процедура поиска называется «линейным пробированием», потому что она основана на линейной проверке последовательности ячеек.

Приложение Hash Workshop

Приложение Hash Workshop демонстрирует метод линейного пробирования. На рис. 11.5 показано, как выглядит рабочая область приложения при запуске.

В приложении значения ключей лежат в диапазоне от 0 до 999. Исходный размер массива равен 60. Хеш-функция должна сжать диапазон ключей до размера массива. Для этого, как уже упоминалось ранее, применяется оператор вычисления остатка (`%`):

```
arrayIndex = key % arraySize;
```

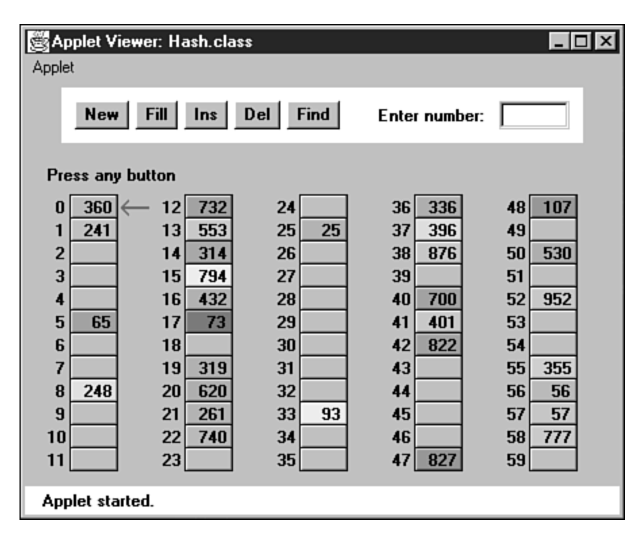


Рис. 11.5. Приложение Hash Workshop при запуске

Для исходного размера массива 60 получается:

```
arrayIndex = key % 60;
```

Хеш-функция достаточно проста для того, чтобы вычисления можно было выполнять «в уме». Вычитайте из ключа по 60 до тех пор, пока не получится число, меньшее 60. Например, чтобы вычислить хеш-код 143, вычтите 60 (получается 83), а затем еще 60 (получается 23). Результат определяет индекс ячейки, в которой алгоритм разместит значение 143. Вы можете легко проверить правильность хеширования. (Для массива из 10 ячеек проверка выполняется еще проще — индекс определяется последней цифрой ключа.)

Как и в других приложениях, операции выполняются повторными нажатиями соответствующей кнопки. Например, чтобы найти элемент данных с заданным номером, следует щелкать на кнопке `Find`. Помните, что перед использованием другой кнопки необходимо полностью отработать всю серию нажатий с одной кнопкой. Например, не переключайтесь с нажатий `Fill` на другую кнопку, пока не появится сообщение *Press any key*.

Выполнение любых операций начинается с ввода числового значения. Например, для кнопки `Find` необходимо ввести значение ключа, а для кнопки `New` — размер новой таблицы.

Кнопка New

Кнопка `New` создает новую хеш-таблицу заданного размера. Максимальный размер таблицы равен 60; ограничение определяется количеством ячеек, помещающихся в окне приложения. Размер по умолчанию тоже равен 60. Мы используем это число, потому что оно упрощает проверку правильности хеш-кодов, но как будет показано позднее, в хеш-таблицах общего назначения размер массива должен быть простым числом, поэтому значение 59 было бы более уместным.

Кнопка Fill

Изначально хеш-таблица содержит 30 элементов, то есть она заполнена только наполовину. Кнопка `Fill` позволяет заполнить ее заданным количеством элементов данных. Щелкните на кнопке `Fill` и введите количество генерируемых элементов. Хеш-таблицы лучше всего работают, когда они заполнены не более чем на половину или максимум на две трети (40 элементов в таблице из 60 ячеек).

Заполненные ячейки распределяются по таблице неравномерно. Иногда несколько пустых ячеек следуют подряд, иногда встречаются серии заполненных ячеек.

Назовем последовательность заполненных ячеек в хеш-таблице *заполненной последовательностью*. По мере добавления новых элементов заполненные последовательности становятся все длиннее. Это явление, называемое *группировкой*, продемонстрировано на рис. 11.6.

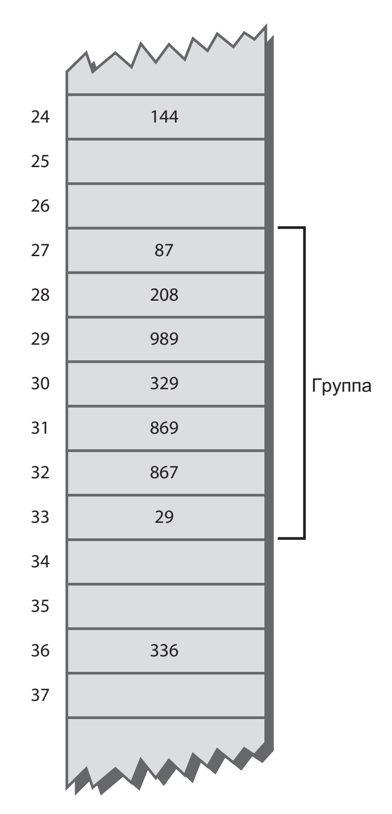


Рис. 11.6. Пример группировки

Учтите, что при работе с приложением операция заполнения с почти заполненной таблицей (например, попытка размещения 59 элементов в таблице из 60 ячеек) выполняется относительно долго. Если вам покажется, что программа «зависла» — будьте терпеливы. Заполнение почти заполненного массива в высшей степени неэффективно.

Также обратите внимание на то, что при заполнении хеш-таблицы все алгоритмы перестают работать; приложение предполагает, что в таблице имеется хотя бы одна пустая ячейка.

Кнопка Find

Кнопка `Find` применяет хеш-функцию к значению ключа, введенному в текстовом поле. Вычисление дает индекс в массиве. Если ячейка с этим индексом свободна, это оптимальный вариант, и метод может немедленно сообщить об успешном выполнении операции.

Однако может оказаться, что ячейка уже занята элементом данных с другим ключом. В таком случае происходит коллизия; занятая ячейка помечается красной стрелкой. Алгоритм поиска переходит

к следующей ячейке в последовательности. Процесс поиска подходящей ячейки после коллизии называется *пробированием*.

При возникновении коллизии алгоритм `Find` последовательно перебирает каждую ячейку последовательности. Обнаружив пустую ячейку до того, как будет обнаружен искомый ключ, он знает, что поиск завершился неудачей. Искать дальше бессмысленно, потому что алгоритм вставки поместил бы элемент в эту (или более раннюю) ячейку. На рис. 11.7 изображены успешные и неуспешные попытки линейного пробирования.

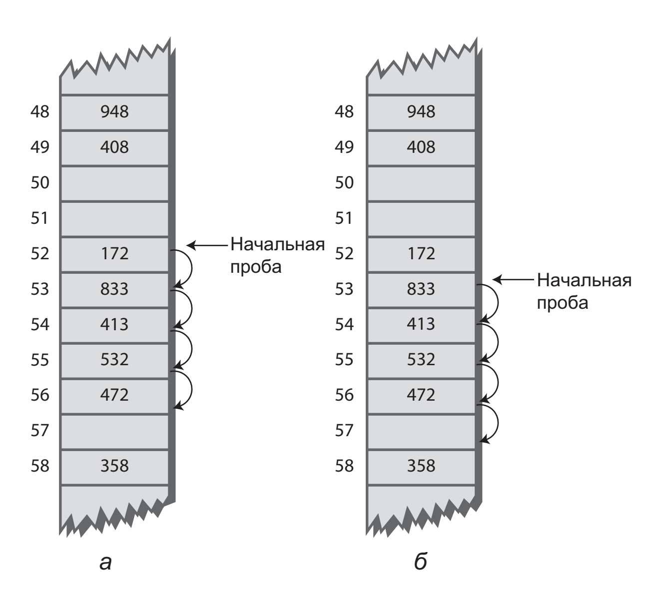


Рис. 11.7. Линейное пробирование: *а* — успешный поиск элемента 472; *б* — безуспешный поиск элементв 893

Кнопка Ins

Кнопка `Ins` вставляет в хеш-таблицу элемент данных с ключом, введенным в текстовом поле. При вставке используется тот же алгоритм, как и при поиске кнопкой `Find`. Если начальная ячейка занята, алгоритм осуществляет линейный поиск свободной ячейки. Обнаружив ячейку, алгоритм вставляет элемент.

Попробуйте вставить несколько новых элементов данных. Введите число из трех цифр и понаблюдайте за происходящим. Большинство элементов размещается в первой же опробованной ячейке, но время от времени возникают коллизии, и алгоритму приходится перебирать ячейки в поисках свободной. Количество проверяемых при этом ячеек называется *длиной пробирования*. Как правило, она составляет всего несколько ячеек, но в отдельных случаях длина пробирования может составить 4 или 5 ячеек и даже больше при крайне высокой степени заполнения массива.

Обратите внимание на то, какие ключи хешируются в одинаковые значения индекса. Если размер массива равен 60, то ключи 7, 67, 127, 187, 247 и т. д. до 967

хешируются в индекс 7. Попробуйте вставить в таблицу эту или аналогичную последовательность — это наглядно продемонстрирует процесс линейного пробирования.

Кнопка Del

Кнопка Del удаляет элемент с ключом, введенным пользователем. Удаление не может быть реализовано простым удалением данных из ячейки. Почему? Вспомните, что в процессе вставки процесс пробирования перебирает серию ячеек в поисках свободного места. Если в середине этой последовательности заполненных ячеек неожиданно появится пустая ячейка, то процедура поиска сочтет ее признаком неудачи, хотя при продолжении перебора нужная ячейка была бы в конечном итоге найдена.

По этой причине удаленный элемент заменяется элементом со специальным значением ключа, которое идентифицирует его как удаленный. Наше приложение считает, что все допустимые значения ключей положительны, поэтому для пометки удаленных элементов можно воспользоваться значением –1. Удаленные элементы помечаются специальным маркером *Del*.

Кнопка Ins вставляет новый элемент в первую обнаруженную пустую ячейку или в ячейку с элементом *Del*. Кнопка Find интерпретирует элементы *Del* как существующие элементы для принятия решений о продолжении поиска.

Если количество удалений относительно велико, хеш-таблица быстро заполняется эрзац-элементами *Del*, что снижает ее эффективность. По этой причине многие реализации хеш-таблиц не поддерживают удаление. А если такая возможность существует, ей не следует злоупотреблять.

Дубликаты

Может ли хеш-таблица содержать элементы данных с одинаковыми ключами? Метод заполнения в приложении Hash Workshop не допускает наличия дубликатов, но при желании их можно вставить при помощи кнопки Ins. Вы увидите, что доступ ограничивается первым экземпляром дубликата, а обращение ко второму возможно только после удаления первого. Согласитесь, это не слишком удобно.

Алгоритм Find можно переписать таким образом, чтобы он искал все элементы с совпадающими ключами, не ограничиваясь первым. Но для этого придется проводить поиск по всем ячейкам каждой линейной последовательности. Это приведет к потерям времени при любых обращениях к таблицам — даже тех, в которых дубликаты отсутствуют. Вероятно, на практике в большинстве случаев дубликаты лучше запретить.

Группировка

Попробуйте вставить в хеш-таблицу приложения Hash Workshop дополнительные элементы. По мере заполнения группы становятся все длиннее. Группировка приводит к значительному увеличению продолжительности пробирования. А это означает, что обращения к ячейкам в конце последовательности происходят очень медленно.

Чем сильнее заполняется массив, тем серьезнее становится проблема группировки. В массиве, заполненном наполовину, эта проблема обычно не проявляется; и даже при заполнении массива на две трети все еще приемлемо. Но превышение этого порога приводит к серьезному снижению быстродействия, так как группы становятся все больше и больше.

Следовательно, при проектировании хеш-таблицы очень важно позаботиться о том, чтобы таблица никогда не заполнялась более чем наполовину или по крайней мере на две трети. (Математическая связь между заполнением хеш-таблицы и длиной пробирования будет рассматриваться позднее в этой главе.)

Реализация хеш-таблицы с линейным пробированием на языке Java

Написать методы поиска, вставки и удаления для хеш-таблиц с линейным пробированием не так уж сложно. Сначала мы рассмотрим реализацию этих методов на языке Java, а затем — полный код программы `hash.java`, в которой эти методы используются в практическом контексте.

Метод find()

Метод `find()` сначала вызывает `hashFunc()`, чтобы хешировать искомый ключ для получения индекса `hashVal`. Метод `hashFunc()` применяет оператор `%` к искомому ключу и размеру массива, как было показано ранее.

Затем в условии `while` метод `find()` проверяет, пуста ли ячейка с данным индексом (`null`). Если ячейка не пуста, метод проверяет, содержит ли она искомый ключ. Если проверка дает положительный результат, то `find()` возвращает элемент, а если нет — увеличивает `hashVal` и возвращается к началу цикла `while`, чтобы проверить, занята ли следующая ячейка.

Программный код `find()` выглядит так:

```
public DataItem find(int key)    // Поиск элемента с заданным ключом
// (Метод предполагает, что таблица не заполнена)
   {
   int hashVal = hashFunc(key);  // Хеширование ключа

   while(hashArray[hashVal] != null)  // Пока не будет найдена пустая ячейка
      {                               // Ключ найден?
      if(hashArray[hashVal].getKey() == key)
         return hashArray[hashVal];   // Да, вернуть элемент
      ++hashVal;                 // Переход к следующей ячейке
      hashVal %= arraySize;      // При достижении конца таблицы
      }                          // происходит возврат к началу
   return null;                  // Элемент не найден
   }
```

Перебирая ячейки массива, `hashVal` в конечном итоге доберется до конца. Когда это произойдет, перебор должен вернуться к началу. Для этого можно было бы

воспользоваться командой if, обнуляющей переменную hashVal при достижении размера массива. Однако для достижения того же эффекта проще применить оператор % к hashVal и размеру массива.

Осторожный программист не станет полагаться на то, что таблица еще не заполнена, как это сделано в нашем примере. Допускать заполнения таблицы нельзя, потому что в этом случае цикл в методе будет выполняться бесконечно. Мы не проверяем это условие ради простоты кода.

Метод insert()

Приведенный ниже метод insert() использует похожий алгоритм для определения позиции, в которой должен находиться элемент данных, но вместо конкретного элемента он ищет пустую ячейку или удаленный элемент (ключ –1). Обнаружив пустую ячейку, insert() размещает в ней новый элемент.

```
public void insert(DataItem item) // Вставка элемента данных
// (Метод предполагает, что таблица не заполнена)
   {
   int key = item.getKey();      // Получение ключа
   int hashVal = hashFunc(key);  // Хеширование ключа
                                          // Пока не будет найдена
   while(hashArray[hashVal] != null &&    // пустая ячейка или -1,
                           hashArray[hashVal].iData != -1)
      {
      ++hashVal;                 // Переход к следующей ячейке
      hashVal %= arraySize;      // При достижении конца таблицы
      }                          // происходит возврат к началу
   hashArray[hashVal] = item;    // Вставка элемента
   }
```

Метод delete()

Следующая реализация метода delete() использует код, аналогичный find(), для поиска существующего элемента. Обнаружив элемент, delete() заменяет его специальным элементом данных nonItem с заранее определенным ключом –1.

```
public DataItem delete(int key)  // Удаление элемента данных
   {
   int hashVal = hashFunc(key);  // Хеширование ключа

   while(hashArray[hashVal] != null)  // Пока не будет найдена пустая ячейка
      {                               // Ключ найден?
      if(hashArray[hashVal].getKey() == key)
         {
         DataItem temp = hashArray[hashVal]; // Временное сохранение
         hashArray[hashVal] = nonItem;       // Удаление элемента
         return temp;                        // Метод возвращает элемент
         }
      ++hashVal;                 // Переход к следующей ячейке
```

```
      hashVal %= arraySize;       // При достижении конца таблицы
      }                           // происходит возврат к началу
   return null;                   // Элемент не найден
   }
```

Программа hash.java

В листинге 11.1 приведен полный код программы hash.java. В этой программе объект DataItem, представляющий элемент данных, содержит всего одно поле типа int — ключ. Как и в других структурах данных, рассмотренных ранее, такие объекты также могут содержать другие поля или ссылку на объект другого класса (например, employee или partNumber).

Важнейшее поле класса HashTable — hashArray — содержит массив ячеек. В других полях хранится текущий размер массива и специальный объект nonItem, используемый для удаления.

Листинг 11.1. Программа hash.java

```
// hash.java
// Реализация хеш-таблицы с линейным пробированием
// Запуск программы: C:>java HashTableApp
import java.io.*;
////////////////////////////////////////////////////////////////
class DataItem
   {                                // (Данных может быть и больше)
   private int iData;               // Данные (ключ)
//--------------------------------------------------------------
   public DataItem(int ii)          // Конструктор
      { iData = ii; }
//--------------------------------------------------------------
   public int getKey()
      { return iData; }
//--------------------------------------------------------------
   }  // Конец класса DataItem
////////////////////////////////////////////////////////////////
class HashTable
   {
   private DataItem[] hashArray;    // Массив ячеек хеш-таблицы
   private int arraySize;
// -------------------------------------------------------------
   public HashTable(int size)       // Конструктор
      {
      arraySize = size;
      hashArray = new DataItem[arraySize];
      nonItem = new DataItem(-1);   // Ключ удаленного элемента -1
      }
// -------------------------------------------------------------
   public void displayTable()
      {
      System.out.print("Table: ");
```

```
      for(int j=0; j<arraySize; j++)
         {
         if(hashArray[j] != null)
            System.out.print(hashArray[j].getKey() + " ");
         else
            System.out.print("** ");
         }
      System.out.println("");
      }
// -------------------------------------------------------------
   public int hashFunc(int key)
      {
      return key % arraySize;       // Хеш-функция
      }
// -------------------------------------------------------------
   public void insert(DataItem item) // Вставка элемента данных
      // (Метод предполагает, что таблица не заполнена)
      {
      int key = item.getKey();      // Получение ключа
      int hashVal = hashFunc(key);  // Хеширование ключа
                                       // Пока не будет найдена
      while(hashArray[hashVal] != null &&  // пустая ячейка или -1,
                     hashArray[hashVal].getKey() != -1)
         {
         ++hashVal;                 // Переход к следующей ячейке
         hashVal %= arraySize;      // При достижении конца таблицы
         }                          // происходит возврат к началу
      hashArray[hashVal] = item;    // Вставка элемента
      }
// -------------------------------------------------------------
   public DataItem delete(int key)  // Удаление элемента данных
      {
      int hashVal = hashFunc(key);  // Хеширование ключа

      while(hashArray[hashVal] != null)   // Пока не будет найдена
                                          // пустая ячейка
         {                                // Ключ найден?
         if(hashArray[hashVal].getKey() == key)
            {
            DataItem temp = hashArray[hashVal]; // Временное сохранение
            hashArray[hashVal] = nonItem;       // Удаление элемента
            return temp;                        // Метод возвращает элемент
            }
         ++hashVal;                 // Переход к следующей ячейке
         hashVal %= arraySize;      // При достижении конца таблицы
         }                          // происходит возврат к началу
      return null;                  // Элемент не найден
      }
// -------------------------------------------------------------
```

продолжение ⇒

Листинг 11.1 (*продолжение*)

```
   public DataItem find(int key)    // Поиск элемента с заданным ключом
      // (Метод предполагает, что таблица не заполнена)
      {
      int hashVal = hashFunc(key);  // Хеширование ключа

      while(hashArray[hashVal] != null)  // Пока не будет найдена
                                         // пустая ячейка
         {                               // Ключ найден?
         if(hashArray[hashVal].getKey() == key)
            return hashArray[hashVal];   // Да, вернуть элемент
         ++hashVal;                 // Переход к следующей ячейке
         hashVal %= arraySize;      // При достижении конца таблицы
         }                          // происходит возврат к началу
      return null;                  // Элемент не найден
      }
// -------------------------------------------------------------
}  // Конец класса HashTable
////////////////////////////////////////////////////////////////
class HashTableApp
   {
   public static void main(String[] args) throws IOException
      {
      DataItem aDataItem;
      int aKey, size, n, keysPerCell;
                                    // Ввод размеров
      System.out.print("Enter size of hash table: ");
      size = getInt();
      System.out.print("Enter initial number of items: ");
      n = getInt();
      keysPerCell = 10;
                                    // Создание таблицы
      HashTable theHashTable = new HashTable(size);

      for(int j=0; j<n; j++)        // Вставка данных
         {
         aKey = (int)(java.lang.Math.random() *
                                    keysPerCell * size);
         aDataItem = new DataItem(aKey);
         theHashTable.insert(aDataItem);
         }

      while(true)                   // Взаимодействие с пользователем
         {
         System.out.print("Enter first letter of ");
         System.out.print("show, insert, delete, or find: ");
         char choice = getChar();
         switch(choice)
            {
            case 's':
               theHashTable.displayTable();
```

```
            break;
         case 'i':
            System.out.print("Enter key value to insert: ");
            aKey = getInt();
            aDataItem = new DataItem(aKey);
            theHashTable.insert(aDataItem);
            break;
         case 'd':
            System.out.print("Enter key value to delete: ");
            aKey = getInt();
            theHashTable.delete(aKey);
            break;
         case 'f':
            System.out.print("Enter key value to find: ");
            aKey = getInt();
            aDataItem = theHashTable.find(aKey);
            if(aDataItem != null)
               {
               System.out.println("Found " + aKey);
               }
            else
               System.out.println("Could not find " + aKey);
            break;
         default:
            System.out.print("Invalid entry\n");
         }
      }
   }
//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   public static char getChar() throws IOException
      {
      String s = getString();
      return s.charAt(0);
      }
//-------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
//--------------------------------------------------------------
   }  // Конец класса HashTableApp
////////////////////////////////////////////////////////////////
```

Метод `main()` класса `HashTableApp` реализует пользовательский интерфейс с возможностями вывода содержимого хеш-таблицы (команда *s*), вставки (*i*), удаления (*d*) или поиска элемента (*f*).

Сначала программа предлагает пользователю ввести размер хеш-таблицы и количество элементов в ней. Хеш-таблица может иметь любой размер, от десятка элементов до 10 000. (На построение таблиц большего размера потребуется некоторое время.) Не используйте команду *s* для таблиц, содержащих более нескольких сотен элементов; вывод содержимого займет слишком много времени.

Переменная `keysPerCell` в методе `main()` задает отношение размеров диапазона ключей и массива. В листинге ей присваивается значение 10. Это означает, что для таблицы с размером 20 значения ключей лежат в диапазоне от 0 до 200.

Если вы хотите разобраться в том, что происходит в программе, создайте таблицу примерно с 20 элементами, чтобы все они помещались в одной строке. Пример взаимодействия с программой `hash.java`:

```
Enter size of hash table: 12
Enter initial number of items: 8

Enter first letter of show, insert, delete, or find: s
Table: 108 13 0 ** ** 113 5 66 ** 117 ** 47

Enter first letter of show, insert, delete, or find: f
Enter key value to find: 66
Found 66

Enter first letter of show, insert, delete, or find: i
Enter key value to insert: 100
Enter first letter of show, insert, delete, or find: s
Table: 108 13 0 ** 100 113 5 66 ** 117 ** 47

Enter first letter of show, insert, delete, or find: d
Enter key value to delete: 100
Enter first letter of show, insert, delete, or find: s
Table: 108 13 0 ** -1 113 5 66 ** 117 ** 47
```

Значения ключей лежат в диапазоне от 0 до 119 (12*10 – 1). Знак ** означает, что ячейка пуста. Элемент с ключом 100 вставляется в ячейку 4 (нумерация начинается с 0), потому что 100%12 = 4. Обратите внимание на то, как при удалении элемента 100 заменяется значением –1.

Расширение массива

Когда хеш-таблица становится слишком полной, один из возможных вариантов заключается в расширении массива. В языке Java массивы имеют фиксированный размер и не могут расширяться динамически. Программа должна создать новый массив большего размера, а затем вставить в него все содержимое старого (меньшего) массива.

Напомним, что хеш-функция при вычислении местонахождения заданного элемента данных учитывает размер массива, поэтому в больших и малых массивах

элементы должны занимать разные ячейки. Следовательно, элементы нельзя просто скопировать из одного массива в другой. Придется последовательно, ячейка за ячейкой, перебрать старый массив и вставить каждый найденный элемент в новый массив методом insert(). Процесс *перехеширования* занимает много времени, но он необходим для расширения массива.

Расширенный массив обычно вдвое больше исходного массива. Вообще говоря, поскольку размер массива должен быть простым числом, размер нового массива должен возрасти немного более, чем вдвое. Вычисление нового размера массива является частью процесса перехеширования.

Ниже приведена пара вспомогательных методов, которые пригодятся для вычисления нового размера массива (или исходного размера массива, если вы не уверены в том, что пользователь введет простое число). Алгоритм начинает с заданного размера и ищет первое простое число, большее заданного. Метод getPrime() находит следующее простое число, большее своего аргумента. Для проверки каждого числа, превышающего заданный размер, вызывается метод isPrime().

```
private int getPrime(int min)   // Возвращает первое простое число > min
   {
   for(int j = min+1; true; j++)    // Для всех j > min
      if( isPrime(j) )              // Число j простое?
         return j;                  // Да, вернуть его
   }
// -------------------------------------------------------------
private boolean isPrime(int n)      // Число n простое?
   {
   for(int j=2; (j*j <= n); j++)    // Для всех j
      if( n % j == 0)               // Делится на j без остатка?
         return false;              // Да, число не простое
   return true;                     // Нет, число простое
   }
```

Код методов весьма примитивен. Например, в getPrime() можно было бы проверить 2, а затем только нечетные числа (вместо того, чтобы проверять каждое число). Однако такие усовершенствования несущественны, потому что для обнаружения простого числа обычно хватает нескольких проверок.

В языке Java имеется класс Vector для представления векторов — структуры, аналогичной массиву, но с возможностью динамического расширения. Однако этот класс особой пользы не принесет, так как изменение размера таблицы требует перехеширования всех элементов данных.

Квадратичное пробирование

Итак, при открытой адресации с линейным пробированием возникает проблема группировки. Образовавшиеся группы начинают расширяться. Элементы, хешируемые в пределах группы, добавляются в конец группы, в результате чего группа становится еще больше. Чем больше размер группы, тем быстрее она растет. Явление напоминает собирающуюся толпу зевак: первые зрители подходят посмотреть,

что случилось, а остальные хотят узнать, на что все смотрят. Чем больше толпа, тем больше людей она привлекает.

Отношение количества элементов в таблице к размеру таблицы называется *коэффициентом заполнения*. Таблица с 10 000 ячеек, содержащая 6667 элементов, имеет коэффициент заполнения 2/3.

```
коэффициент_заполнения = количество_элементов / размер_массива;
```

Группы могут образовываться даже при относительно небольшом коэффициенте заполнения. Одни части хеш-таблицы могут быть заполнены большими группами, другие почти не содержать элементов. Группировка снижает быстродействие таблицы.

Квадратичное пробирование пытается избежать образования групп. Его идея заключается в том, чтобы проверять ячейки, находящиеся на больших расстояниях (вместо ячеек, находящихся вблизи от исходной позиции хеширования).

Вычисление шага

При линейном пробировании, если первичный индекс хеширования равен x, то последующие пробы проверяют позиции $x + 1$, $x + 2$, $x + 3$ и т. д. При квадратичном пробировании проверяются позиции $x + 1$, $x + 4$, $x + 9$, $x + 16$, $x + 25$ и т. д. Расстояние от исходной позиции вычисляется как квадрат номера шага: $x + 1^2$, $x + 2^2$, $x + 3^2$, $x + 4^2$, $x + 5^2$ и т. д.

Пример квадратичного пробирования представлен на рис. 11.8.

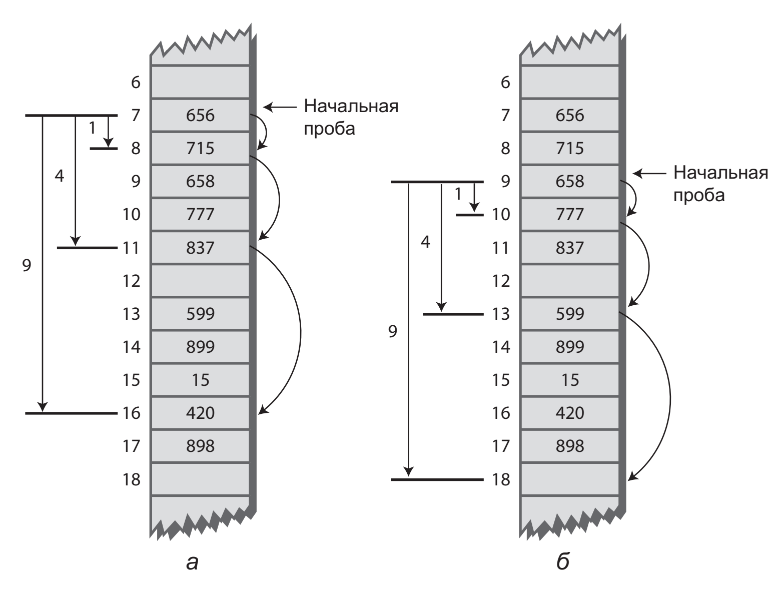


Рис. 11.8. Квадратичное пробирование: *а* — успешный поиск элемента 420; *б* — безуспешный поиск элемента 481

Чтобы понять, как работает квадратичное пробирование, запустите приложение и создайте новую хеш-таблицу с 59 элементами кнопкой New. Когда вам будет пред-

ложено выбрать способ пробирования, выберите квадратичное (`Quad`). Заполните только что созданную таблицу на 4/5 кнопкой `Fill` (47 элементов в массиве из 59 ячеек). Степень заполнения превышает рекомендуемый порог, но с ней будут генерироваться более длинные серии проб, чтобы вы могли изучить алгоритм пробирования.

Если хеш-таблица окажется слишком полной, приложение может вывести сообщение *Can't complete fill* — это происходит при слишком длинных последовательностях пробирования. Каждый дополнительный шаг в процессе пробирования увеличивает размер шага. Если последовательность получается слишком длинной, размер шага в конечном итоге превысит разрядность целочисленной переменной; чтобы этого не произошло, приложение прерывает процесс заполнения.

Когда таблица будет заполнена, выберите одно из существующих значений ключа, нажмите клавишу `Find` и посмотрите, найден ли его алгоритм. Нередко искомый ключ обнаруживается в первой же ячейке или следующей за ней. Но при некотором терпении вы найдете ключ, поиск которого требует трех или четырех шагов, и увидите, как смещение увеличивается с каждым шагом. Также кнопка `Find` может использоваться для поиска несуществующего ключа; поиск продолжается до тех пор, пока при переборе не будет обнаружена пустая ячейка.

ПОЛЕЗНЫЙ СОВЕТ

Важно: следите за тем, чтобы размер массива был простым числом. Например, используйте 59 вместо 60. (Другие простые числа — 53, 47, 43, 41, 37, 31, 29, 23, 19, 17, 13, 11, 7, 5, 3 и 2.) Если размер массива не является простым числом, в ходе пробирования возможна бесконечная последовательность проверок. Если это произойдет в ходе операции заполнения, приложение «зависнет».

Проблемы с квадратичным пробированием

Квадратичное пробирование решает проблему группировки, присущую линейной группировке; эта разновидность группировки называется *первичной*. Однако при квадратичном пробировании возникает другая, более тонкая проблема группировки. Дело в том, что все ключи, хешируемые в конкретный индекс, ищут свободную ячейку в одной и той же последовательности.

Допустим, элементы 184, 302, 420 и 544 хешируются в индекс 7 и вставляются в хеш-таблицу в указанном порядке. В этом случае индекс 302 потребует смещения на одну ячейку, индекс 420 — смещения на 4 ячейки, а индекс 544 — на 9 ячеек. У каждого дополнительного элемента, хешируемого в индекс 7, смещение будет еще большим. Это явление называется *вторичной группировкой*.

Вторичная группировка не создает серьезных проблем. И все же квадратичное пробирование на практике применяется нечасто, потому что существует другое, более удачное решение.

Двойное хеширование

Для устранения как первичной, так и вторичной группировки применяется алгоритм *двойного хеширования*. Вторичная группировка возникает из-за того, что

алгоритм, генерирующий последовательность смещений для квадратичного пробирования, всегда генерирует одни и те же смещения: 1, 4, 9, 16 и т. д.

В идеале последовательность проб должна генерироваться в зависимости от ключа (вместо использования набора одинаковых смещений для всех ключей). В этом случае числа с разными ключами, хешируемые в один индекс, будут использовать разные последовательности смещений.

Задача решается повторным хешированием ключа с другой хеш-функцией и использованием результата в качестве смещения. Для заданного ключа размер смещения остается постоянным при пробировании, но для разных ключей используются разные размеры.

Практический опыт показал, что вторичная хеш-функция должна обладать некоторыми характеристиками:

- Она не должна совпадать с первичной хеш-функцией.
- Ее результат никогда не должен быть равен 0 (в противном случае смещения не будет, все пробы будут приходиться на одну ячейку, а алгоритм войдет в бесконечный цикл).

Эксперты обнаружили, что для решения этой задачи хорошо подходят функции вида

```
смещение = константа - (ключ % константа);
```

где *константа* — простое число, меньшее размера массива. Пример функции:

```
stepSize = 5 - (key % 5);
```

Эта вторичная хеш-функция используется в приложении HashDouble Workshop. Разные ключи могут хешироваться в один индекс, но для них (вероятно) будут сгенерированы разные смещения. При такой хеш-функции размеры смещений лежат в диапазоне от 1 до 5 (рис. 11.9).

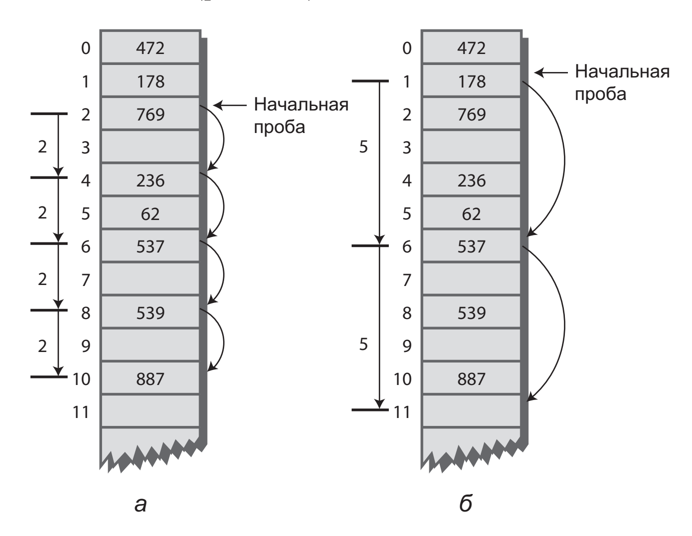


Рис. 11.9. Двойное хеширование: *а* — успешный поиск элемента 887; *б* — безуспешный поиск элемента 709

Двойное хеширование в приложении HashDouble

Приложение HashDouble Workshop показывает, как работает двойное хеширование. Автоматически оно запускается в режиме двойного хеширования, но если позднее переключить его в режим квадратичного пробирования, для возврата к двойному хешированию следует создать новую таблицу кнопкой New и выбрать режим Double, когда вам будет предложено это сделать. Двойное хеширование нагляднее всего демонстрируется на примере таблицы, заполненной на 9/10 емкости и выше. Даже при таких высоких коэффициентах заполнения большинство элементов данных будет немедленно обнаруживаться первым применением хеш-функции; лишь немногие элементы потребуют дополнительных проб.

Попробуйте найти элементы с существующими ключами. Когда поиск потребует применения смещений, вы увидите, что все смещения для заданного ключа имеют одинаковый размер, но для разных ключей выбираются разные смещения (от 1 до 5).

Реализация двойного хеширования на языке Java

В листинге 11.2 приведен полный код программы hashDouble.java, использующей двойное хеширование. Эта программа похожа на программу hash.java (листинг 11.1), но в ней используются две хеш-функции: для поиска начального индекса и для генерирования смещения. Как и прежде, пользователь может вывести текущее содержимое таблицы, вставлять, удалять или проводить поиск элементов.

Листинг 11.2. Программа hashDouble.java

```
// hashDouble.java
// Реализация хеш-таблицы с двойным хешированием
// Запуск программы: C:>java HashDoubleApp
import java.io.*;
////////////////////////////////////////////////////////////////
class DataItem
   {                                 // (Данных может быть и больше)
   private int iData;                // Данные (ключ)
//--------------------------------------------------------------
   public DataItem(int ii)           // Конструктор
      { iData = ii; }
//--------------------------------------------------------------
   public int getKey()
      { return iData; }
//--------------------------------------------------------------
   }  // Конец класса DataItem
////////////////////////////////////////////////////////////////
class HashTable
   {
   private DataItem[] hashArray;     // Массив ячеек хеш-таблицы
   private int arraySize;
   private DataItem nonItem;         // Для удаленных элементов
```

продолжение ⇨

Листинг 11.2 (*продолжение*)

```
// -------------------------------------------------------------
   HashTable(int size)              // Конструктор
      {
      arraySize = size;
      hashArray = new DataItem[arraySize];
      nonItem = new DataItem(-1);
      }
// -------------------------------------------------------------
   public void displayTable()
      {
      System.out.print("Table: ");
      for(int j=0; j<arraySize; j++)
         {
         if(hashArray[j] != null)
            System.out.print(hashArray[j].getKey()+ " ");
         else
            System.out.print("** ");
         }
      System.out.println("");
      }
// -------------------------------------------------------------
   public int hashFunc1(int key)
      {
      return key % arraySize;
      }
// -------------------------------------------------------------
   public int hashFunc2(int key)
      {
      // Возвращаемое значение отлично от нуля, меньше размера массива,
      // функция отлична от хеш-функции 1
      // Размер массива должен быть простым по отношению к 5, 4, 3 и 2
      return 5 - key % 5;
      }
// -------------------------------------------------------------
                                    // Вставка элемента данных
   public void insert(int key, DataItem item)
   // (Метод предполагает, что таблица не заполнена)
      {
      int hashVal = hashFunc1(key);  // Хеширование ключа
      int stepSize = hashFunc2(key); // Вычисление смещения
                                          // Пока не будет найдена
      while(hashArray[hashVal] != null &&    // пустая ячейка или -1
                        hashArray[hashVal].getKey() != -1)
         {
         hashVal += stepSize;        // Прибавление смещения
         hashVal %= arraySize;       // Возврат к началу
         }
      hashArray[hashVal] = item;     // Вставка элемента
      }
```

```
// -------------------------------------------------------------
   public DataItem delete(int key)   // Удаление элемента данных
      {
      int hashVal = hashFunc1(key);      // Хеширование ключа
      int stepSize = hashFunc2(key);     // Вычисление смещения

      while(hashArray[hashVal] != null)  // Пока не найдена пустая ячейка
         {                               // Ключ найден?
         if(hashArray[hashVal].getKey() == key)
            {
            DataItem temp = hashArray[hashVal]; // Временное сохранение
            hashArray[hashVal] = nonItem;       // Удаление элемента
            return temp;                        // Метод возвращает элемент
            }
         hashVal += stepSize;            // Прибавление смещения
         hashVal %= arraySize;           // Возврат к началу
         }
      return null;                   // Элемент не найден
      }
// -------------------------------------------------------------
   public DataItem find(int key)     // Поиск элемента с заданным ключом
   // (Метод предполагает, что таблица не заполнена)
      {
      int hashVal = hashFunc1(key);      // Хеширование ключа
      int stepSize = hashFunc2(key);     // Вычисление смещения

      while(hashArray[hashVal] != null)  // Пока не найдена пустая ячейка
         {                               // Ключ найден?
         if(hashArray[hashVal].getKey() == key)
            return hashArray[hashVal];   // Да, метод возвращает элемент
         hashVal += stepSize;            // Прибавление смещения
         hashVal %= arraySize;           // Возврат к началу
         }
      return null;                   // Элемент не найден
      }
// -------------------------------------------------------------
   }  // Конец класса HashTable
////////////////////////////////////////////////////////////////
class HashDoubleApp
   {
   public static void main(String[] args) throws IOException
      {
      int aKey;
      DataItem aDataItem;
      int size, n;
                               // Ввод размеров
      System.out.print("Enter size of hash table: ");
      size = getInt();
      System.out.print("Enter initial number of items: ");
```

продолжение ⇨

Листинг 11.2 (*продолжение*)

```
      n = getInt();
                                  // Создание таблицы
      HashTable theHashTable = new HashTable(size);

      for(int j=0; j<n; j++)      // Вставка данных
         {
         aKey = (int)(java.lang.Math.random() * 2 * size);
         aDataItem = new DataItem(aKey);
         theHashTable.insert(aKey, aDataItem);
         }

      while(true)                 // Взаимодействие с пользователем
         {
         System.out.print("Enter first letter of ");
         System.out.print("show, insert, delete, or find: ");
         char choice = getChar();
         switch(choice)
            {
            case 's':
               theHashTable.displayTable();
               break;
            case 'i':
               System.out.print("Enter key value to insert: ");
               aKey = getInt();
               aDataItem = new DataItem(aKey);
               theHashTable.insert(aKey, aDataItem);
               break;
            case 'd':
               System.out.print("Enter key value to delete: ");
               aKey = getInt();
               theHashTable.delete(aKey);
               break;
            case 'f':
               System.out.print("Enter key value to find: ");
               aKey = getInt();
               aDataItem = theHashTable.find(aKey);
               if(aDataItem != null)
                  System.out.println("Found " + aKey);
               else
                  System.out.println("Could not find " + aKey);
               break;
            default:
               System.out.print("Invalid entry\n");
            }
         }
      }
//-------------------------------------------------------------
```

```
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//-------------------------------------------------------------
   public static char getChar() throws IOException
      {
      String s = getString();
      return s.charAt(0);
      }
//-------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
//-------------------------------------------------------------
   }  // Конец класса HashDoubleApp
////////////////////////////////////////////////////////////////
```

Выходные данные и принципы работы этой программы почти не отличаются от hash.java. В таблице 11.1 показано, что произойдет при вставке 21 элемента в хеш-таблицу из 23 ячеек с применением двойного хеширования. Смещения изменяются в диапазоне от 1 до 5.

Таблица 11.1. Заполнение таблицы из 23 ячеек с применением двойного хеширования

Номер	**Ключ**	**Хеш-код**	**Смещение**	**Последовательность проб**
1	1	1	4	
2	38	15	2	
3	37	14	3	
4	16	16	4	
5	20	20	5	
6	3	3	2	
7	11	11	4	
8	24	1	1	2
9	5	5	5	
10	16	16	4	20 1 5 9
11	10	10	5	

продолжение ⇗

Таблица 11.1 (*продолжение*)

Номер	Ключ	Хеш-код	Смещение	Последовательность проб
12	31	8	4	
13	18	18	2	
14	12	12	3	
15	30	7	5	
16	1	1	4	
17	19	19	1	
18	36	13	4	17
19	41	18	4	22
20	15	15	5	20 2 7 12 17 22 4
21	25	2	5	7 12 17 22 4 9 14 19 1 6

Первые 15 ключей хешируются в основном в свободные ячейки (10-й ключ — исключение). Далее по мере заполнения массива последовательности проб удлиняются. Итоговый массив ключей выглядит так:

```
** 1 24 3 15 5 25 30 31 16 10 11 12 1 37 38 16 36 18 19 20 ** 41
```

Выбор простого размера таблицы

При использовании двойного хеширования размер таблицы должен быть простым числом. Чтобы понять смысл этого требования, представьте ситуацию, в которой размер таблицы простым числом не является. Предположим, размер массива равен 15 (индексы от 0 до 14), а конкретный ключ хешируется в исходный индекс 0 со смещением 5. Пробы будут выполняться в последовательности 0, 5, 10, 0, 5, 10 и т. д. до бесконечности. Проверяются только эти три ячейки, поэтому алгоритм «не увидит» пустые ячейки 1, 2, 3 и т. д., а из попытки выполнения операции ничего не выйдет.

Если бы размер массива был равен 13 (простое число), то в процессе пробирования в конечном итоге была бы проверена каждая ячейка: 0, 5, 10, 2, 7, 12, 4, 9, 1, 6, 11, 3 и т. д. Даже если в массиве имеется всего одна пустая ячейка, она будет успешно обнаружена. Простой размер массива не делится нацело ни на какое число, поэтому последовательность проб рано или поздно проверит каждую ячейку.

Аналогичный эффект проявляется с квадратичным пробированием. Однако в этом случае смещение увеличивается с каждым шагом, что в конечном итоге приведет к переполнению переменной, в которой оно хранится (хотя это тоже приведет к прерыванию бесконечного цикла).

В общем случае при использовании открытой адресации предпочтение следует отдавать двойному хешированию.

Метод цепочек

При открытой адресации коллизии разрешаются поиском свободных ячеек в хеш-таблице. Другое возможное решение основано на ведении отдельного связанного списка по каждому индексу в хеш-таблице. Ключ элемента данных хешируется в индекс обычным способом, а полученный элемент вставляется в связанный список по этому индексу. Другие элементы, хешируемые в тот же индекс, просто добавляются в связанный список; искать пустые ячейки в первичном массиве просто не нужно. Принцип работы метода цепочек показан на рис. 11.10.

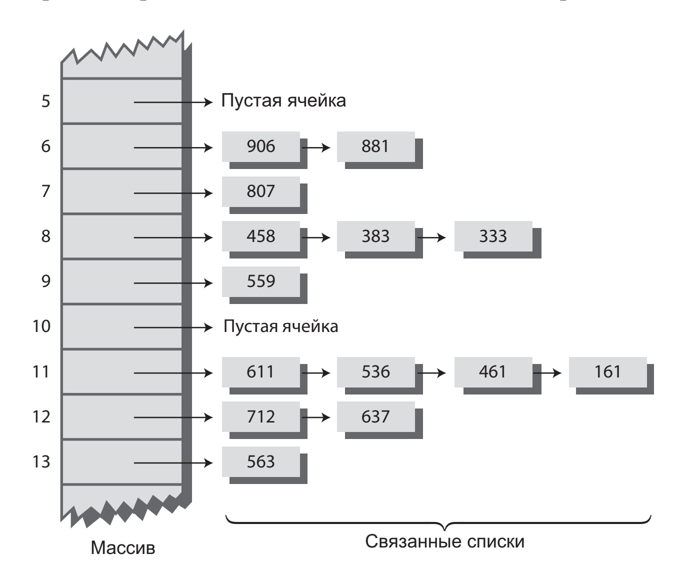


Рис. 11.10. Метод цепочек

На концептуальном уровне метод цепочек реализуется проще, чем различные схемы пробирования, используемые при открытой адресации. С другой стороны, объем кода увеличивается, так как программа должна реализовать механизм ведения связанных списков — как правило, для их представления в программу вводятся дополнительные классы.

Приложение HashChain Workshop

Приложение HashChain Workshop демонстрирует принципы работы хеширования с методом цепочек. В рабочей области приложения отображается содержимое массива связанных списков (рис. 11.11).

Каждый элемент массива занимает одну строку, а связанные списки выводятся слева направо. Изначально массив содержит 25 ячеек (25 списков). На экране все эти данные не помещаются; чтобы увидеть все содержимое массива, прокрутите изображение вверх или вниз. В приложении выводятся до 6 элементов каждого списка. Вы можете создать хеш-таблицу, содержащую до 100 списков, и использо-

вать коэффициенты заполнения до 2,0. При более высоких значениях количество элементов в связанных списках может превысить 6; лишние элементы выходят за правый край, и пользователь не сможет просмотреть все содержимое списков. (Это периодически происходит даже с коэффициентом заполнения 2,0.)

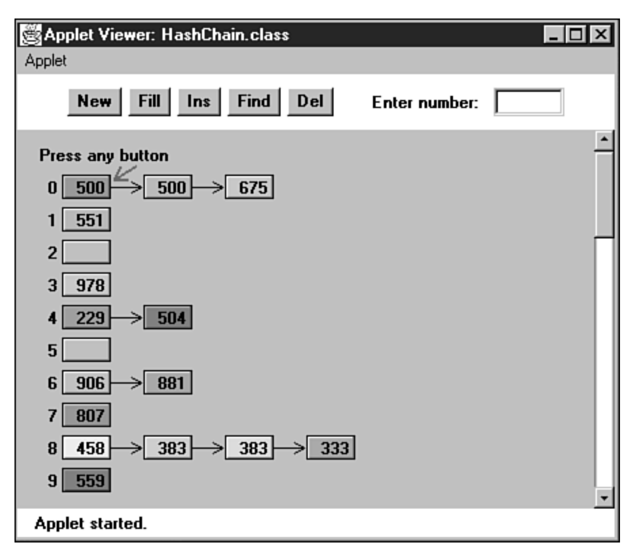


Рис. 11.11. Приложение HashChain Workshop

Поэкспериментируйте с приложением HashChain, попробуйте вставить новые элементы при помощи кнопки `Ins`. Вы увидите, как красная стрелка немедленно переходит к нужному списку, в начало которого вставляется элемент. Списки в приложении HashChain не отсортированы, поэтому вставка не требует поиска по списку. (Работа с сортированными списками продемонстрирована в программе `hashChain.java`.)

Проверьте, как работает поиск элементов с использованием кнопки `Find`. Если при выполнении операции окажется, что ячейка списка содержит несколько элементов, красная стрелка перебирает их в поисках нужного. Как было показано в главе 5, «Связанные списки», успешный поиск требует просмотра приблизительно половины элементов списка. При безуспешном поиске проверяются все элементы списка.

Коэффициенты заполнения

Коэффициент заполнения (отношение количества элементов в хеш-таблице к ее размеру) при использовании метода цепочек обычно отличается от коэффициента заполнения при открытой адресации. В методе цепочек массив из *N* ячеек вполне может содержать более *N* элементов; таким образом, коэффициент заполнения может быть равен единице и более. Это абсолютно нормально; коэффициент, больший единицы, попросту означает, что списки некоторых ячеек состоят из двух и более элементов.

Конечно, большое количество элементов в списках замедляет обращение к данным, потому что для обращения к заданному элементу приходится просматривать в среднем половину элементов списка. Исходная ячейка находится быстро за время $O(1)$, но поиск по списку выполняется за время, пропорциональное M — среднему количеству элементов в списке. Следовательно, поиск выполняется за время $O(M)$, а чрезмерное заполнение списков нежелательно.

Как показывает приложение Workshop, хеш-таблицы с коэффициентом заполнения 1 встречаются достаточно часто. При таком коэффициенте заполнения примерно треть ячеек остается пустой, треть содержит один элемент и еще треть — два и более элементов.

При открытой адресации коэффициент заполнения, превышающий порог в 1/2 или 2/3, приводит к резкому снижению быстродействия. При реализации метода цепочек коэффициент заполнения может превысить единицу без сколько-нибудь значительного ущерба для быстродействия. Таким образом, метод цепочек обладает значительно большей стабильностью, особенно если объем данных в хеш-таблице трудно предсказать заранее.

Дубликаты

Дубликаты разрешены и могут создаваться в процессе заполнения `Fill`. Все элементы с одинаковыми ключами вставляются в один список; чтобы вернуть их все, и успешный, и неуспешный поиск требует перебора всего списка, что отражается на быстродействии. Операция `Find` в приложении находит только первый экземпляр элемента с заданным ключом.

Удаление

При использовании метода цепочек удаление создает меньше проблем, чем при открытой адресации. Алгоритм хеширует ключ в нужный индекс, а затем удаляет элемент из списка. Так как пробирование не используется, список в конкретной ячейке может стать пустым — это ни на что не повлияет. Кнопка `Del` в приложении Workshop демонстрирует работу удаления.

Размер таблицы

С методом цепочек выбор простых чисел в качестве размера таблицы уже не играет такой важной роли, как при квадратичном пробировании и двойном хешировании. Метод цепочек обходится без проб, поэтому беспокоиться о возможном зацикливании процесса пробирования (из-за размера массива, кратного смещению) уже не нужно.

С другой стороны, некоторые виды распределения ключей могут привести к группировке данных, когда размер массива не является простым числом. Эта проблема будет более подробно рассмотрена при обсуждении хеш-функций.

Гнезда

Другое решение, концептуально сходное с методом цепочек, заключается в создании массива (вместо связанного списка) в каждой ячейке хеш-таблицы. Такие массивы иногда называются *гнездами* (buckets). Однако гнездовые решения по эффективности уступают решениям на базе связанных списков из-за проблемы выбора размера гнездовых массивов. Слишком малые размеры массивов могут привести к переполнению, а слишком большие — к неэффективному расходованию памяти. У связанных списков с динамическим выделением памяти такой проблемы не существует.

Реализация метода цепочек на языке Java

В программу `hashChain.java` включен класс `SortedList` и вспомогательный класс `Link`. Сортировка списков не ускоряет успешный поиск, но время безуспешного поиска сокращается примерно вдвое. (Решение о неудаче принимается сразу же при достижении элемента, большего искомого ключа, что в среднем занимает половину элементов массива.) Время удаления тоже сокращается вдвое; с другой стороны, время вставки увеличивается, потому что новый элемент нельзя просто вставить в начало списка; перед вставкой необходимо определить его место в упорядоченном списке. Если списки остаются достаточно короткими, замедление вставки может оказаться несущественным.

Если при работе с хеш-таблицей прогнозируется большое количество безуспешных попыток поиска, возможно, вместо неупорядоченного списка стоит воспользоваться более сложным упорядоченным списком. Но если скорость вставки играет более важную роль, предпочтение отдается неупорядоченному списку.

Программа `hashChain.java` (листинг 11.3) сначала строит хеш-таблицу с параметрами, заданными пользователем (размер и количество элементов). Далее пользователь может выполнять операции вставки, поиска, удаления и вывода содержимого списка. Чтобы вся хеш-таблица помещалась на экране, ее размер не должен быть более 16 или около того.

Листинг 11.3. Программа hashChain.java

```
// hashChain.java
// Реализация хеш-таблицы с использованием метода цепочек
// Запуск программы: C:>java HashChainApp
import java.io.*;
////////////////////////////////////////////////////////////////
class Link
   {                                   // (Здесь могут быть другие поля)
   private int iData;                  // Данные
   public Link next;                   // Следующий элемент списка
// -------------------------------------------------------------
   public Link(int it)                 // Конструктор
      { iData= it; }
// -------------------------------------------------------------
```

```
   public int getKey()
      { return iData; }
// -------------------------------------------------------------
   public void displayLink()           // Вывод содержимого элемента
      { System.out.print(iData + " "); }
   }  // Конец класса Link
////////////////////////////////////////////////////////////////
class SortedList
   {
   private Link first;               // Ссылка на первый элемент списка
// -------------------------------------------------------------
   public void SortedList()          // Конструктор
      { first = null; }
// -------------------------------------------------------------
   public void insert(Link theLink)  // Вставка в порядке сортировки
      {
      int key = theLink.getKey();
      Link previous = null;          // Начиная с первого элемента
      Link current = first;
                                     // До конца списка
      while( current != null && key > current.getKey() )
         {                           // Или пока current <= key
         previous = current;
         current = current.next;     // Переход к следующему элементу
         }
      if(previous==null)             // В начале списка
         first = theLink;            // first --> новый элемент
      else                           // Не в начале
         previous.next = theLink;    // prev --> новый элемент
      theLink.next = current;        // новый элемент --> current
      }
// -------------------------------------------------------------
   public void delete(int key)       // Удаление элемента
      {                              // (предполагается, что список не пуст)
      Link previous = null;          // Начиная с первого элемента
      Link current = first;
                                     // До конца списка
      while( current != null && key != current.getKey() )
         {                           // или пока key != current
         previous = current;
         current = current.next;     // Переход к следующему элементу
         }
                                     // Отсоединение
      if(previous==null)             // Если первый элемент,
         first = first.next;         // изменить first
      else                           // В противном случае
         previous.next = current.next; // удалить текущий элемент
      }  // end delete()
// -------------------------------------------------------------
```

продолжение ⇨

Листинг 11.3 (*продолжение*)

```
   public Link find(int key)         // Поиск элемента с заданным ключом
      {
      Link current = first;          // Начиная от начала списка
                                     // До конца списка
      while(current != null &&  current.getKey() <= key)
         {                           // или пока ключ не превысит current,
         if(current.getKey() == key)    // Искомый элемент?
            return current;          // Совпадение обнаружено
         current = current.next;     // Переход к следующему элементу
         }
      return null;                   // Элемент не найден
      }
// -------------------------------------------------------------
   public void displayList()
      {
      System.out.print("List (first-->last): ");
      Link current = first;       // От начала списка
      while(current != null)      // Перемещение до конца списка
         {
         current.displayLink();   // Вывод данных
         current = current.next;  // Переход к следующему элементу
         }
      System.out.println("");
      }
   }  // Конец класса SortedList
////////////////////////////////////////////////////////////////
class HashTable
   {
   private SortedList[] hashArray;   // Массив списков
   private int arraySize;
// -------------------------------------------------------------
   public HashTable(int size)        // Конструктор
      {
      arraySize = size;
      hashArray = new SortedList[arraySize];  // Создание массива
      for(int j=0; j<arraySize; j++)          // Заполнение массива
         hashArray[j] = new SortedList();     // списками
      }
// -------------------------------------------------------------
   public void displayTable()
      {
      for(int j=0; j<arraySize; j++) // Для каждой ячейки
         {
         System.out.print(j + ". "); // Вывод номера ячейки
         hashArray[j].displayList(); // Вывод списка
         }
      }
// -------------------------------------------------------------
```

```
   public int hashFunc(int key)      // Хеш-функция
      {
      return key % arraySize;
      }
// -------------------------------------------------------------
   public void insert(Link theLink)  // Вставка элемента
      {
      int key = theLink.getKey();
      int hashVal = hashFunc(key);   // Хеширование ключа
      hashArray[hashVal].insert(theLink); // Вставка в позиции hashVal
      }
// -------------------------------------------------------------
   public void delete(int key)       // Удаление элемента
      {
      int hashVal = hashFunc(key);   // Хеширование ключа
      hashArray[hashVal].delete(key); // Удаление
      }
// -------------------------------------------------------------
   public Link find(int key)         // Поиск элемента
      {
      int hashVal = hashFunc(key);   // Хеширование ключа
      Link theLink = hashArray[hashVal].find(key);  // Поиск
      return theLink;                // Метод возвращает найденный элемент
      }
// -------------------------------------------------------------
   }  // Конец класса HashTable
////////////////////////////////////////////////////////////////
class HashChainApp
   {
      public static void main(String[] args) throws IOException
      {
      int aKey;
      Link aDataItem;
      int size, n, keysPerCell = 100;
                                    // Ввод размеров
      System.out.print("Enter size of hash table: ");
      size = getInt();
      System.out.print("Enter initial number of items: ");
      n = getInt();
                                 // Создание таблицы
      HashTable theHashTable = new HashTable(size);

      for(int j=0; j<n; j++)         // Вставка данных
         {
         aKey = (int)(java.lang.Math.random() *
                                    keysPerCell * size);
         aDataItem = new Link(aKey);
         theHashTable.insert(aDataItem);
         }
```

продолжение ☞

Листинг 11.3 (*продолжение*)

```
      while(true)                    // Взаимодействие с пользователем
         {
         System.out.print("Enter first letter of ");
         System.out.print("show, insert, delete, or find: ");
         char choice = getChar();
         switch(choice)
            {
            case 's':
               theHashTable.displayTable();
               break;
            case 'i':
               System.out.print("Enter key value to insert: ");
               aKey = getInt();
               aDataItem = new Link(aKey);
               theHashTable.insert(aDataItem);
               break;
            case 'd':
               System.out.print("Enter key value to delete: ");
               aKey = getInt();
               theHashTable.delete(aKey);
               break;
            case 'f':
               System.out.print("Enter key value to find: ");
               aKey = getInt();
               aDataItem = theHashTable.find(aKey);
               if(aDataItem != null)
                  System.out.println("Found " + aKey);
               else
                  System.out.println("Could not find " + aKey);
                  break;
            default:
               System.out.print("Invalid entry\n");
            }
         }
      }
//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//-------------------------------------------------------------
   public static char getChar() throws IOException
      {
      String s = getString();
      return s.charAt(0);
      }
```

```
//-------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
      }
//--------------------------------------------------------------
   }  // Конец класса HashChainApp
////////////////////////////////////////////////////////////////
```

В следующем примере пользователь создает таблицу с 20 списками, вставляет в нее 20 элементов и выводит ее содержимое командой *s*:

```
Enter size of hash table: 20
Enter initial number of items: 20
Enter first letter of show, insert, delete, or find: s
0. List (first-->last): 240 1160
1. List (first-->last):
2. List (first-->last):
3. List (first-->last): 143
4. List (first-->last): 1004
5. List (first-->last): 1485 1585
6. List (first-->last):
7. List (first-->last): 87 1407
8. List (first-->last):
9. List (first-->last): 309
10. List (first-->last): 490
11. List (first-->last):
12. List (first-->last): 872
13. List (first-->last): 1073
14. List (first-->last): 594 954
15. List (first-->last): 335
16. List (first-->last): 1216
17. List (first-->last): 1057 1357
18. List (first-->last): 938 1818
19. List (first-->last):
```

При увеличении количества элементов в таблице вы увидите, что длина списков растет, но элементы остаются отсортированными. Также предусмотрена возможность удаления элементов.

Мы еще вернемся к вопросу использования метода цепочек при обсуждении эффективности хеш-таблиц (см. далее в этой главе).

Хеш-функции

В этом разделе мы разберемся, какими свойствами должна обладать хорошая хеш-функция, а также попробуем усовершенствовать метод хеширования строк, описанный в начале этой главы.

Быстрые вычисления

Хорошая хеш-функция должна быть простой, чтобы ее результат быстро вычислялся. Главное преимущество хеш-таблиц — скорость. Если хеш-функция работает медленно, то это преимущество будет утрачено. Хеш-функция с многочисленными операциями умножения и деления вряд ли хорошо справится со своим делом. (Хотя иногда в этом помогают поразрядные операции Java и C++ — например, сдвиг числа вправо эквивалентен делению на степень 2.)

Основной целью хеш-функции является преобразование диапазона ключей в диапазон индексов, обеспечивающее равномерное распределение ключей по индексам хеш-таблицы. Ключи могут быть как полностью случайными, так и частично детерминированными.

Случайные ключи

Идеальная хеш-функция ставит в соответствие каждому значению ключа уникальный индекс. Такое поведение возможно только при нетипично «хорошем» поведении ключей с диапазоном, достаточно узким для прямого использования в качестве индексов (как в примере с табельными номерами работников в начале главы).

На практике обычно ни одно из этих условий не выполняется, и хеш-функции приходится сжимать широкий диапазон ключей в меньший диапазон индексов.

Желательное поведение хеш-функции зависит от распределения значений ключей в конкретной базе данных. Ранее мы предполагали, что данные равномерно распределены по всему диапазону. Для такой ситуации хеш-функция вида

```
индекс = ключ % размер_массива;
```

достаточно хорошо работает. В ней используется только одна математическая операция, поэтому для действительно случайных ключей полученные индексы тоже будут случайными, а следовательно, будут иметь хорошее распределение.

Неслучайные ключи

Однако на практике данные часто распределяются не полностью случайным образом. Представьте базу данных, в которой в качестве ключей используются номера деталей машин — допустим, числа в формате

033-400-03-94-05-0-535

Формат номера интерпретируется следующим образом:

Цифры 0–2: код поставщика (от 1 до 999, в настоящее время до 70).
Цифры 3–5: код категории (100, 150, 200, 250 и т. д. до 850).
Цифры 6–7: месяц запуска в производство (от 1 до 12).
Цифры 8–9: год запуска в производство (от 00 до 99).
Цифры 10–11: серийный номер (от 1 до 99).
Цифра 12: признак токсичности (0 или 1).

Цифры 13–15: контрольная сумма (остаток от деления суммы других полей на 100).

Ключ для приведенного номера детали будет иметь вид 0 334 000 394 050 535. Однако такие ключи распределены неравномерно. Большинство чисел из диапазона 0 до 9 999 999 999 999 999 не может использоваться в качестве ключа (например, числа с кодами поставщиков выше 70, коды категорий, не кратные 50, или месяцы от 13 до 99). Кроме того, контрольная сумма вычисляется в зависимости от других чисел. Чтобы эти числа образовали более случайный диапазон, их необходимо подвергнуть дополнительной обработке.

Исключение неинформативных частей

Поля ключа необходимо по возможности «сжать», чтобы каждый их бит нес полезную информацию. Например, поле кода категории можно изменить так, чтобы в нем могли храниться только значения в диапазоне от 0 до 15. Кроме того, поле контрольной суммы следует исключить из ключа, так как оно не содержит дополнительной информации; это избыточные данные, намеренно введенные с целью контроля. Для сжатия полей ключа применяются различные поразрядные операции.

Использование всех данных

Каждая часть ключа (кроме неинформативных частей, о которых говорилось выше) должна вносить свой вклад в вычисление хеш-функции. Не ограничивайтесь первыми четырьмя цифрами или другими аналогичными схемами. Чем больше данных учитывается при вычислении ключа, тем больше вероятность хеширования ключей по всему диапазону индексов.

Иногда диапазон допустимых значений ключей оказывается настолько большим, что он приводит к переполнению переменных типа `int` и `long`. О том, как решается проблема переполнения, будет рассказано ниже, когда мы вернемся к хешированию строк.

Подведем итог: хеш-функция должна быть простой и быстрой. При этом она должна исключать из обработки неинформативные части ключа и использовать все данные.

Использование простых чисел при вычислении остатка

В хеш-функциях часто используется оператор вычисления остатка (`%`) с размером таблицы. Вы уже знаете, что выбор простого числа в качестве размера таблицы играет важную роль при квадратичном пробировании и двойном хешировании. Но если сами ключи имеют неслучайное распределение, размер таблицы должен быть простым числом независимо от выбора системы хеширования.

Дело в том, что если многие ключи имеют общий делитель с размером массива, они часто хешируются в одну позицию, а это приводит к группировке. Простой размер таблицы исключает такую возможность. Например, если размер таблицы

был бы кратен 50 в нашем примере с деталями машин, то все коды категорий хешировались бы в индексы, кратные 50. Простое число (например, 53) гарантирует, что ключи не будут делиться нацело на размер таблицы.

Итак, тщательно анализируйте ключи и адаптируйте алгоритм хеширования для устранения любых аномалий в распределении ключей.

Хеширование строк

В начале главы рассматривался способ преобразования коротких строк в числовые ключи, основанный на умножении кодов символов на степени константы. Например, в соответствии с этой схемой слово *cats* преобразовывалось в число по формуле

```
ключ = 3*273 + 1*272 + 20*271 + 19*270
```

К достоинствам такого решения следует отнести то, что в нем задействованы все символы входной строки. Вычисленное значение ключа хешируется в индекс массива обычным способом:

```
index = (key) % arraySize;
```

Метод Java для вычисления ключа по строке выглядит так:

```
public static int hashFunc1(String key)
   {
   int hashVal = 0;
   int pow27 = 1;                        // 1, 27, 27*27 и т. д.

   for(int j=key.length()-1; j>=0; j--) // Справа налево
      {
      int letter = key.charAt(j) - 96;  // Получение кода символа
      hashVal += pow27 * letter;        // Умножение на степень 27
      pow27 *= 27;                      // Следующая степень 27
      }
   return hashVal % arraySize;
   }
```

Цикл начинается с последней буквы слова. Если слово состоит из N букв, то последняя имеет индекс $N - 1$. В переменной `letter` сохраняется числовой эквивалент буквы в кодировке, определенной нами в начале главы ($a = 1$ и т. д.) Затем код умножается на степень 27: единица для буквы в позиции $N - 1$, 27 для буквы в позиции $N - 2$ и т. д.

Метод `hashFunc1()` не так эффективен, как хотелось бы. Помимо преобразования символов, в цикле выполняются два умножения и сложение. Чтобы исключить умножение, можно воспользоваться математической формулой, называемой *методом Горнера* (английский математик, 1773–1827). Согласно этой формуле, выражение вида

$$var4*n^4 + var3*n^3 + var2*n^2 + var1*n^1 + var0*n^0$$

может быть записано в виде

```
(((var4*n + var3)*n + var2)*n + var1)*n + var0
```

Вычисление результата по этой формуле начинается с внутренних скобок и двигается наружу. В результате преобразования формулы в код Java мы получаем следующий метод:

```
public static int hashFunc2(String key)
   {
   int hashVal = key.charAt(0) - 96;
   for(int j=1; j<key.length(); j++)    // Слева направо
      {
      int letter = key.charAt(j) - 96;  // Получение кода символа
      hashVal = hashVal * 27 + letter;  // Умножение и сложение
      }
   return hashVal % arraySize;          // Вычисление остатка
   }
```

На этот раз мы начинаем с первой, а не с последней буквы слова (что более естественно), а при каждой итерации цикла выполняется одно умножение и одно сложение (не считая извлечения символа из строки).

К сожалению, метод `hashFunc2()` не может работать со строками, длина которых превышает семь символов. Превышение пороговой длины приводит к тому, что значение `hashVal` выходит за границы типа `int`. (При использовании типа `long` та же проблема возникает со строками большей длины).

Можно ли изменить сам базовый алгоритм, чтобы избежать переполнения переменных? Обратите внимание: ключ, получаемый в результате хеширования, всегда меньше размера массива, потому что мы применяем оператор получения остатка. Слишком велик не итоговый индекс, а промежуточные значения ключа.

Формула Горнера позволяет включить оператор `%` в каждый шаг вычисления. В результате мы получим такой же результат, как при однократном применении оператора в конце, но избежим промежуточного переполнения. (При этом в цикле добавляется новая операция.) Реализация приведена в методе `hashFunc3()`:

```
public static int hashFunc3(String key)
   {
   int hashVal = 0;
   for(int j=0; j<key.length(); j++)    // Слева направо
      {
      int letter = key.charAt(j) - 96;  // Получение кода символа
      hashVal = (hashVal * 27 + letter) % arraySize; // Оператор %
      }
   return hashVal;                      // Без оператора %
   }
```

Этот способ (или его аналоги) обычно применяют при хешировании строк. Также можно воспользоваться различными приемами, связанными с поразрядными операциями со строками — например, использованием базы 32 (или большей степени 2) вместо 27, чтобы умножение можно было выполнить посредством оператора >>, выполняемого быстрее оператора `%`.

Аналогичный метод может использоваться для преобразования произвольных строк в число, подходящее для хеширования. Строки могут содержать числа, имена или результаты конкатенации любых подстрок.

Свертка

Другая разумная хеш-функция основана на разбиении ключа на группы цифр с последующим суммированием групп. Такое решение гарантирует, что хеш-код будет зависеть от каждой цифры исходных данных. Количество цифр в группе должно соответствовать размеру массива. Другими словами, для массива из 1000 элементов каждая группа должна состоять из трех цифр.

Предположим, вы хотите хешировать 9-разрядные номера социального страхования для линейного пробирования. Если размер массива равен 1000, число из 9 цифр делится на три группы из трех цифр. Так, для кода 123-45-6789 вычисляется ключ 123 + 456 + 789 = 1368. Оператор % усекает полученную сумму, чтобы максимальное значение индекса составляло 999. В нашем примере 1368%1000 = 368. Если бы размер массива был равен 100, то ключ из 9 цифр пришлось бы разделить на четыре группы из двух цифр и одну группу из одной цифры: 12 + 34 + 56 + 78 + 9 = 189, и 189%100 = 89.

Когда размер массива кратен 10, работу этой схемы легко понять. Но как было показано для других хеш-функций, оптимальный размер массива должен быть простым числом. Реализация этой схемы оставляется читателю для самостоятельной работы.

Эффективность хеширования

Как упоминалось ранее, эффективность вставки и поиска в хеш-таблицах может достигать $O(1)$. При отсутствии коллизий вставка нового или поиск существующего элемента требует только вызова хеш-функции и одного обращения к элементу массива. Это минимальное время доступа из всех возможных.

При возникновении коллизий время доступа зависит от длины пробирования. Каждое обращение к ячейке в ходе пробирования увеличивает время поиска свободной (вставка) или занятой ячейки (поиск). При каждом обращении необходимо проверить, свободна ли ячейка, или (для поиска или удаления) содержит ли она искомый элемент.

Таким образом, время отдельной операции поиска или вставки пропорционально длине пробирования. Это время прибавляется к постоянному времени вызова хеш-функции.

Средняя длина пробирования (а следовательно, и среднее время обращения) зависит от коэффициента заполнения (отношения количества элементов в таблице к ее размеру). С увеличением коэффициента заполнения увеличивается и длина пробирования.

Рассмотрим связь между длиной пробирования и коэффициентом заполнения для различных видов рассмотренных нами хеш-таблиц.

Открытая адресация

В различных схемах открытой адресации высокий коэффициент заполнения приводит к более заметному снижению эффективности, чем в реализациях метода цепочек.

При открытой адресации безуспешный поиск обычно занимает больше времени, чем успешный. В процессе пробирования алгоритм прекращает перебор сразу же после обнаружения нужного элемента, что в среднем происходит на середине последовательности пробирования. С другой стороны, чтобы удостовериться в отсутствии элемента, алгоритму придется перебрать всю последовательность до самого конца.

Линейное пробирование

Следующие формулы демонстрируют связь между длиной пробирования (P) и коэффициентом заполнения (L) при линейном пробировании. Для успешного поиска длина пробирования вычисляется по формуле

$$P = (1 + 1/(1 - L)^2)/2.$$

А для безуспешного — по формуле

$$P = (1 + 1/(1 - L))/2.$$

Формулы взяты из книги Кнута (см. приложение Б, «Литература»), а их теоретическое обоснование достаточно сложно. На рис. 11.12 представлены графики двух функций.

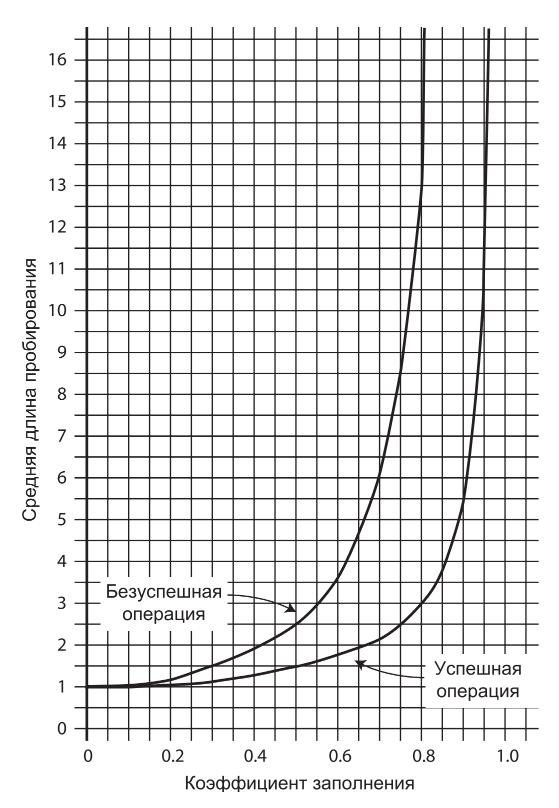


Рис. 11.12. Эффективность линейного пробирования

С коэффициентом заполнения 1/2 успешный поиск требует в среднем 1,5 сравнения, а безуспешный — 2,5 сравнения. При коэффициенте заполнения 2/3 эти показатели составляют 2,0 и 5,0, а с дальнейшим его увеличением они становятся очень большими.

Итак, как видите, коэффициент заполнения должен оставаться меньше 2/3, а желательно — меньше 1/2. С другой стороны, чем ниже коэффициент заполнения, тем больше памяти расходуется на хранение заданного объема данных. Оптимальный коэффициент заполнения для конкретной ситуации определяется балансом между эффективностью использования памяти, снижающейся при низких значениях коэффициента, и скоростью, которая при низких значениях возрастает.

Квадратичное пробирование и двойное хеширование

Эффективность квадратичного пробирования и двойного хеширования определяется одинаковыми формулами, в которых проявляется некоторое преимущество этих методов перед линейным пробированием. Для успешного поиска формула (также взятая из книги Кнута) выглядит так:

$$-\log_2(1 - \text{loadFactor})/\text{loadFactor}.$$

Формула для безуспешного поиска:

$$1/(1 - \text{loadFactor}).$$

Графики этих функций представлены на рис. 11.13. При коэффициенте заполнения 0,5 как успешный, так и безуспешный поиск требуют в среднем двух проб. При коэффициенте 2/3 показатели составляют 2,37 и 3,0 соответственно, а при коэффициенте 0,8 — 2,90 и 5,0. Следовательно, схемы квадратичного пробирования и двойного хеширования лучше переносят увеличение коэффициента заполнения, чем линейное пробирование.

Метод цепочек

Анализ эффективности для метода цепочек несколько отличается от анализа эффективности открытой адресации (и обычно выполняется проще).

Мы хотим знать, сколько времени потребуется для поиска или вставки элемента в хеш-таблицу, реализующую метод цепочек. Будем считать, что самой затратной (по времени) частью этих операций является сравнение искомого ключа с ключами других элементов в списке, что время хеширования и проверки достижения конца списка эквивалентно одному сравнению ключа. В этом случае все операции выполняются за время `1+nComps`, где `nComps` — количество сравнений ключа.

Предположим, хеш-таблица состоит из `arraySize` ячеек, каждая из которых содержит список, и в нее были вставлены `N` элементов данных. В среднем каждый список содержит `N/arraySize` элементов:

```
средняя_длина_списка = N / arraySize
```

Формула совпадает с определением коэффициента заполнения:

```
коэффициент_заполнения = N / arraySize
```

Таким образом, средняя длина списка равна коэффициенту заполнения.

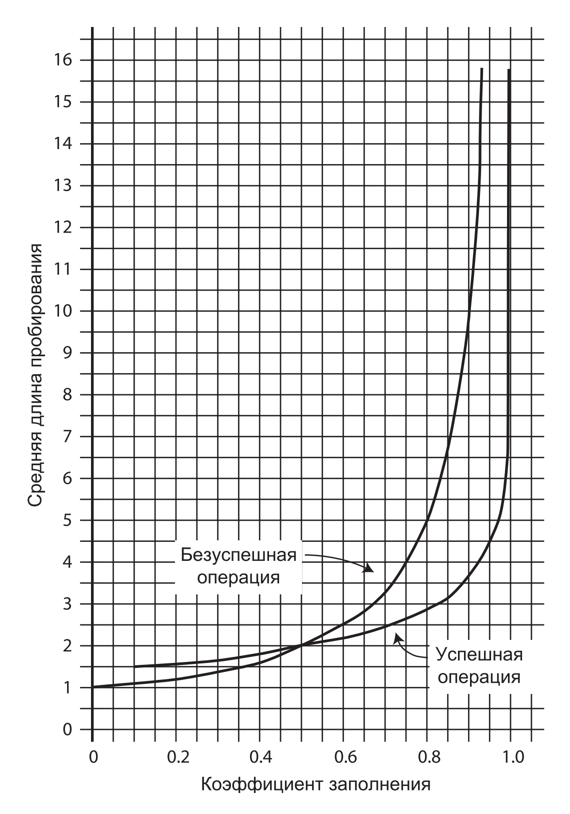


Рис. 11.13. Эффективность квадратичного пробирования и двойного хеширования

Поиск

При успешном поиске алгоритм хеширует ключ в соответствующий список, а затем ищет элемент в списке. В среднем для обнаружения нужного элемента необходимо проверить половину элементов списка. Таким образом, время поиска равно

```
1 +коэффициент_заполнения / 2
```

Формула справедлива как для упорядоченных, так и для неупорядоченных списков. При безуспешном поиске, если списки не упорядочены, приходится проверять все элементы, поэтому время поиска будет равно

```
1 + коэффициент_заполнения
```

Графики этих функций представлены на рис. 11.14.

Для упорядоченного списка безуспешный поиск требует проверки в среднем половины элементов, поэтому время выполнения операции не отличается от времени успешного поиска.

Для метода цепочек характерен коэффициент заполнения около 1,0 (количество элементов данных равно размеру массива). Меньшие значения коэффициента не приводят к значительному улучшению производительности, но время выполнения всех операций возрастает линейно, поэтому превышать значение 2 или около того обычно не рекомендуется.

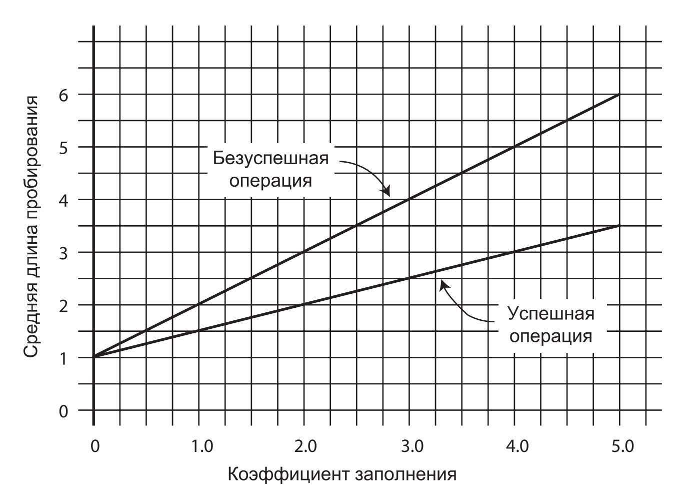


Рис. 11.14. Эффективность метода цепочек

Вставка

В неупорядоченных списках вставка всегда выполняется мгновенно (в том смысле, что она не требует ни одного сравнения). Значение хеш-функции все равно вычислять приходится, поэтому обозначим время вставки 1.

Упорядоченные списки, как и безуспешный поиск, требуют проверки в среднем половины элементов каждого списка, поэтому время вставки равно

1 + коэффициент_заполнения/2.

Сравнение открытой адресации с методом цепочек

При использовании открытой адресации двойное хеширование обычно превосходит квадратичное пробирование. Единственным исключением является ситуация, в которой доступен большой объем памяти, а данные не расширяются после создания таблицы; в этом случае линейное пробирование несколько проще реализуется, а при использовании коэффициентов нагрузки менее 0,5 не приводит к заметным потерям производительности.

Если количество элементов, которые будут вставляться в хеш-таблицу, неизвестно на момент ее создания, метод цепочек предпочтительнее открытой адресации. Увеличение коэффициента заполнения при открытой адресации приводит к резкому снижению быстродействия, а с методом цепочек быстродействие убывает всего лишь линейно.

Если не уверены — используйте метод цепочек. У него есть свои недостатки — прежде всего необходимость в дополнительном классе связанного списка, но зато если в список будет внесено больше данных, чем ожидалось изначально, реализация по крайней мере не замедлится до полного паралича.

Хеширование и внешнее хранение данных

В конце главы 10, «Деревья 2-3-4», рассматривались В-деревья — структуры данных, предназначенные для внешнего хранения данных (на дисковых устройствах). Давайте кратко рассмотрим возможности применения хеш-таблиц для внешнего хранения.

Как упоминалось в главе 10, файл на диске делится на блоки, содержащие сразу несколько записей, а время обращения к блоку значительно превышает время внутренней обработки данных в оперативной памяти. По этим причинам определяющим фактором при разработке стратегии внешнего хранения является минимальное количество обращений к блокам. С другой стороны, внешняя память стоит относительно дешево, поэтому в ней можно хранить много служебной информации, ускоряющей доступ к данным. В решении этой задачи на помощь приходят хеш-таблицы.

Таблица файловых указателей

Центральное место в схеме внешнего хеширования занимает хеш-таблица с номерами блоков во внешней памяти. Иногда эта хеш-таблица называется *индексом*. Она может храниться в оперативной памяти или при слишком больших размерах — на диске, с частичным считыванием ее в оперативную память. Даже если таблица полностью помещается в оперативной памяти, вероятно, ее копия будет храниться на диске и считываться в память при открытии файла.

Неполные блоки

Вернемся к примеру из главы 10: размер блока составляет 8192 байта, а размер записи — 512 байт. Таким образом, в блоке помещается 16 записей. Каждая запись в хеш-таблице указывает на один из этих блоков. Допустим, конкретный файл состоит из 100 блоков. Индекс (хеш-таблица) в оперативной памяти содержит номера блоков файла, от 0 (начало файла) до 99.

При внешнем хешировании важно, чтобы блоки не заполнялись более чем наполовину. Таким образом, блок в среднем должен содержать 8 записей. В одних блоках может быть больше записей, в других — меньше. В среднем файл должен содержать около 800 записей. Структура файла изображена на рис. 11.15.

Все записи с ключами, хешируемыми в один индекс, располагаются в одном блоке. Чтобы найти запись с конкретным ключом, алгоритм поиска хеширует ключ, использует хеш-код как индекс хеш-таблицы, определяет номер блока по указанному индексу, а затем читает блок.

Эффективность такого процесса обусловлена тем, что для обнаружения конкретного элемента достаточно всего одного обращения к блоку. Недостатком являются затраты дискового пространства: значительная часть памяти расходуется впустую, так как частичное заполнение блоков заложено в саму архитектуру.

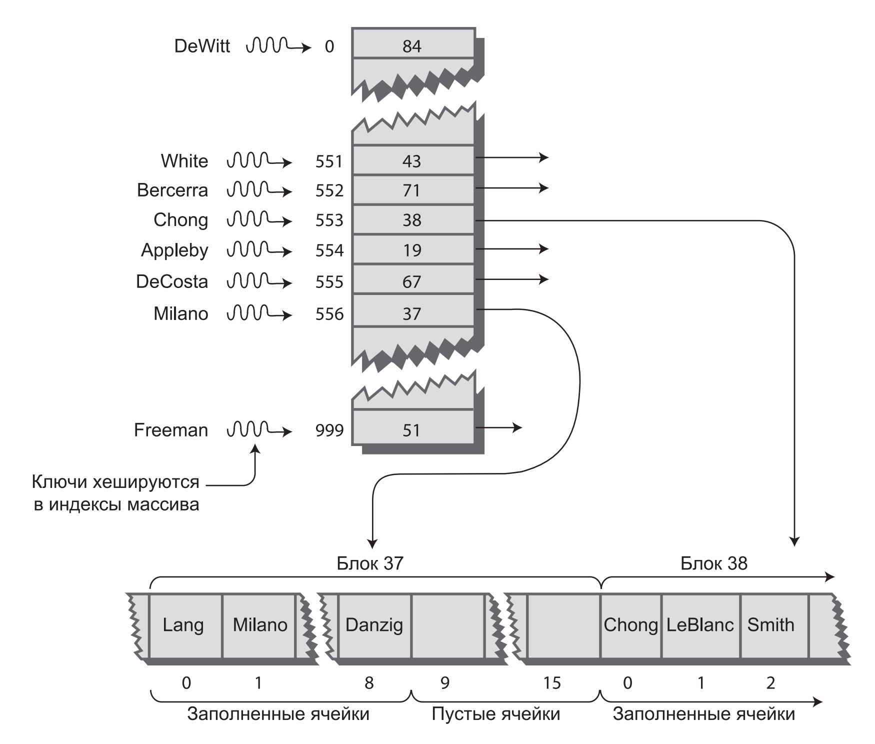


Рис. 11.15. Внешнее хеширование

При реализации этой схемы необходимо тщательно выбрать хеш-функцию и размер хеш-таблицы, чтобы количество ключей, хешируемых в одно значение, оставалось ограниченным. В нашем примере в каждый ключ должны хешироваться в среднем 8 записей.

Полные блоки

Даже с хорошей хеш-функцией блоки время от времени будут заполняться. Проблема решается при помощи видоизмененных схем разрешения коллизий, рассмотренных ранее для внутренних хеш-таблиц: открытой адресации и метода цепочек.

При открытой адресации, если при вставке один блок окажется заполненным, то алгоритм вставит следующую запись в соседний блок. При линейном пробировании это будет следующий блок, но для выбора также может использоваться схема квадратичного пробирования или двойного хеширования. При реализации метода цепочек могут создаваться специальные «блоки переполнения»; если первичный блок окажется заполненным, новая запись вставляется в блок переполнения.

Заполнение блоков нежелательно, потому что для второго блока потребуется лишнее обращение к диску, а это приведет к удвоению времени доступа. Впрочем, если это событие происходит редко, такая схема приемлема.

Мы рассмотрели простейшую реализацию хеш-таблицы для внешнего хранения данных. Существует немало других, более сложных решений, выходящих за рамки книги.

Итоги

- Хеш-таблица создается на базе массива.
- Диапазон значений ключей обычно больше размера массива.
- Хеш-функция преобразует значение ключа в индекс массива.
- Типичным примером базы данных, эффективно реализуемой в форме хеш-таблицы, является словарь английского языка.
- Хеширование ключа в уже заполненную ячейку массива называется коллизией.
- Существует две основные схемы разрешения коллизий: открытая адресация и метод цепочек.
- При открытой адресации элементы данных, хешируемые в заполненную ячейку массива, размещаются в другой ячейке.
- В методе цепочек каждый элемент массива содержит связанный список. Все элементы данных, хешируемые в заданный индекс массива, вставляются в этот список.
- В настоящей главе были рассмотрены три разновидности открытой адресации: линейное пробирование, квадратичное пробирование и двойное хеширование.
- При линейном пробировании смещение всегда равно единице. Таким образом, если вычисленный хеш-функцией индекс массива равен x, то пробирование переходит к ячейкам x, $x + 1$, $x + 2$, $x + 3$ и т. д.
- Количество шагов, необходимых для обнаружения элемента, называется длиной пробирования.
- При линейном пробировании в хеш-таблице появляются непрерывные последовательности заполненных ячеек, называемые первичными группами. Наличие таких групп снижает эффективность хеширования.
- При квадратичном пробировании смещение x равно квадрату номера шага. Таким образом, процесс пробирования проверяет ячейки x, $x + 1$, $x + 4$, $x + 9$, $x + 16$ и т. д.
- Квадратичное пробирование решает проблему первичной группировки, но не избавляет от (менее опасной) вторичной группировки.
- Вторичная группировка возникает из-за того, что для всех ключей, хешируемых в одно значение, в процессе перебора применяется одна и та же последовательность смещений.
- Все ключи, хешируемые в одно значение, следуют по единой последовательности проб, потому что величина смещения не зависит от ключа.

- При двойном хешировании величина смещения зависит от ключа, а для его вычисления используется вторичная хеш-функция.
- Если вторичная хеш-функция возвращает значение s, то процесс пробирования проверяет ячейки x, $x + s$, $x + 2s$, $x + 3s$, $x + 4s$ и т. д. Значение s зависит от ключа, но остается неизменным в процессе пробирования.
- Коэффициентом заполнения называется отношение количества элементов данных в хеш-таблице к размеру массива.
- Максимальный коэффициент заполнения при открытой адресации должен оставаться равным примерно 0,5. Для двойного хеширования с таким коэффициентом заполнения средняя длина пробирования при поиске равна 2.
- Когда коэффициент заполнения при открытой адресации приближается к значению 1,0, время поиска стремится к бесконечности.
- При открытой адресации очень важно, чтобы хеш-таблица не заполнялась выше определенного порога.
- Для метода цепочек приемлем коэффициент заполнения 1,0.
- При таком коэффициенте заполнения успешный поиск в среднем требует 1,5 пробы, а безуспешный поиск — 2,0 пробы.
- Длина пробирования при реализации метода цепочек возрастает линейно с коэффициентом заполнения.
- Хеширование строки может осуществляться умножением кодов символов на разные степени констант, суммированием произведений и применением оператора вычисления остатка (`%`) для сокращения результата до размера хеш-таблицы.
- Чтобы избежать переполнения, можно применять оператор `%` на каждом шаге процесса (представление многочлена по методу Горнера).
- Размер хеш-таблицы обычно должен быть простым числом. Это особенно важно при квадратичном пробировании и методе цепочек.
- Хеш-таблицы могут использоваться для организации внешнего хранения данных. Одно из возможных решений — хеш-таблица, в которой хранятся номера блоков дискового файла.

Вопросы

Следующие вопросы помогут читателю проверить качество усвоения материала. Ответы на них приведены в приложении В.

1. За какое время (в O-синтаксисе) выполняется поиск элемента в хеш-таблице (в идеальном случае)?
2. ___________ преобразует диапазон значений ключей в диапазон значений индексов.

3. Термином «открытая адресация» называется:
   a) наличие в массиве многих свободных ячеек;
   b) свобода выбора адреса;
   c) проверка ячеек $x + 1$, $x + 2$ и т. д., пока не будет найдена пустая ячейка;
   d) поиск другой ячейки в массиве в том случае, если исходная ячейка занята.
4. Использование следующей доступной позиции после безуспешной пробы называется ______________.
5. Какие смещения применяются на первых пяти шагах квадратичного пробирования?
6. Вторичная группировка возникает из-за того, что:
   a) несколько ключей хешируются в одну позицию;
   b) последовательность применяемых смещений всегда одинакова;
   c) в хеш-таблицу вставляется слишком много элементов с одинаковым ключом;
   d) хеш-функция не идеальна.
7. Метод цепочек основан на создании ______________ в каждой ячейке.
8. Приемлемый коэффициент заполнения в методе цепочек равен _________.
9. Возможная хеш-функция для строк основана на умножении кодов символов на последовательно увеличиваемые степени числа (Да/Нет).
10. Если объем данных неизвестен заранее, то оптимальным методом является:
    a) линейное пробирование;
    b) квадратичное пробирование;
    c) двойное хеширование;
    d) метод цепочек.
11. Если в хеш-функции используется метод свертки, то количество цифр в каждой группе должно соответствовать ______________.
12. При линейном пробировании безуспешный поиск занимает больше времени, чем успешный поиск (Да/Нет).
13. В методе цепочек время вставки нового элемента:
    a) возрастает линейно с коэффициентом заполнения;
    b) пропорционально количеству элементов в таблице;
    c) пропорционально количеству списков;
    d) пропорционально доле заполненных ячеек в массиве.
14. При внешнем хешировании важно, чтобы блоки не заполнялись записями.
15. При внешнем хешировании все записи с ключами, хешируемыми в одно значение, располагаются в ____________.

Упражнения

Упражнения помогут вам глубже усвоить материал этой главы. Программирования они не требуют.

1. При линейном пробировании время безуспешного поиска связано с размером групп. Используя приложение Hash Workshop, найдите средний размер группы для 30 элементов в 60 ячейках (коэффициент заполнения 0,5). Изолированная ячейка (то есть ячейка, по обе стороны от которой располагаются свободные ячейки) считается группой размера 1. Чтобы найти среднее значение, можно подсчитать количество ячеек в каждой группе, а затем разделить сумму на количество групп, но существует и более простой способ. Какой? Повторите эксперимент для 5–6 массивов 30/60 и вычислите средний размер групп. Повторите процесс для коэффициентов заполнения 0,6, 0,7, 0,8 и 0,9. Согласуются ли ваши результаты с диаграммой на рис. 11.12?
2. В приложении HashDouble Workshop создайте небольшую хеш-таблицу с квадратичным пробированием, размер которой не является простым числом, — допустим, 24. Заполните ее большим количеством элементов — например, 16. Теперь проведите поиск несуществующих значений ключа. Опробуйте разные ключи, пока не найдете значение, при котором квадратичное пробирование входит в бесконечный цикл. Зацикливание возникает из-за того, что остатки от деления квадратичных смещений на не-простой размер массива образуют повторяющиеся серии. Мораль: размер массива должен быть простым числом.
3. В приложении HashChain создайте массив из 25 ячеек и вставьте в него 50 элементов (коэффициент заполнения 2,0). Проанализируйте связанные списки, отображаемые приложением. Просуммируйте длины всех связанных списков и разделите на их количество, чтобы узнать среднюю длину списка. В среднем безуспешный поиск будет иметь такую длину. (Существует более быстрый способ определения этой средней длины. Что это за способ?)

Программные проекты

В этом разделе читателю предлагаются программные проекты, которые укрепляют понимание материала и демонстрируют практическое применение концепций этой главы.

11.1. Измените программу `hash.java` (см. листинг 11.1) так, чтобы в ней использовалось квадратичное пробирование.

11.2. Реализуйте хеш-таблицу с линейным пробированием, в которой хранятся строки. Вам понадобится хеш-функция, преобразующая строку в индекс; см. раздел «Хеширование строк» этой главы. Предполагается, что строки записаны символами нижнего регистра, поэтому 26 символов английского алфавита должно быть достаточно.

11.3. Напишите хеш-функцию, реализующую метод свертки (см. раздел «Хеш-функции» этой главы). Программа должна работать с произвольным размером

массива и длиной ключа. Используйте линейное пробирование. Обращение к группе цифр числа может быть реализовано проще, чем кажется. Важно ли, чтобы размер массива был кратен 10?

11.4. Напишите метод `rehash()` для программы `hash.java`. Метод должен вызываться из `insert()` для перемещения всей хеш-таблицы в массив примерно вдвое большего размера в ситуациях, когда коэффициент заполнения превышает 0,5. Новый размер массива должен быть простым числом (см. раздел «Расширение массива» этой главы). Не забывайте об обработке «удаленных» элементов (то есть элементов, замененных значением –1).

11.5. Вместо применения связанных списков для разрешения коллизий воспользуйтесь деревом двоичного поиска. Иначе говоря, создайте хеш-таблицу, которая представляет собой массив деревьев. В качестве отправной точки используйте программу `hashChain.java` (см. листинг 11.3) и класс `Tree` из программы `tree.java` (см. листинг 8.1 в главе 8). Для вывода небольшой хеш-таблицы с деревьями можно воспользоваться симметричным обходом деревьев.

Преимущество дерева перед связанным списком заключается в том, что поиск в нем выполняется за время $O(\log N)$ вместо $O(N)$. При высоких коэффициентах заполнения экономия времени может оказаться весьма значительной. Проверка 15 элементов в списке потребует до 15 сравнений, а в дереве — не более 4.

Дубликаты создают проблемы как в деревьях, так и в хеш-таблицах. Добавьте в программу код, который предотвращает вставку дубликатов ключей в хеш-таблицы.

(Будьте внимательны: метод `find()` в классе `Tree` предполагает, что дерево не пусто.) Чтобы сократить объем программного кода, можно отказаться от удаления, которое для деревьев реализуется относительно сложно.

Глава 12

Пирамиды

В главе 4, «Стеки и очереди», упоминалась приоритетная очередь — структура данных, обеспечивающая удобные обращения к элементу данных с наименьшим (или наибольшим) ключом.

Приоритетные очереди применяются для планирования задач в компьютерных областях. Если некоторые программы или операции должны выполняться раньше других, им назначаются более высокие приоритеты.

Другим примером может послужить система управления вооружением — скажем, на крейсере. Система обнаруживает различные угрозы (самолеты, ракеты, подводные лодки и т. д.), которым назначаются разные приоритеты. Так, летящей ракете, находящейся вблизи от крейсера, присваивается более высокий приоритет, чем самолету, находящемуся на большом расстоянии, чтобы средства противодействия (например, зенитные ракеты) были применены к ней в первую очередь.

Приоритетные очереди также используются во внутренней реализации многих компьютерных алгоритмов. В главе 14, «Взвешенные графы», описано применение приоритетных очередей в алгоритмах для работы с графами (в частности, в алгоритме Дейкстры).

Приоритетная очередь представляет собой абстрактный тип данных (ADT) с методами для удаления элемента с наименьшим (или наибольшим) ключом, вставки, а иногда для выполнения других операций. Как и другие ADT, приоритетные очереди могут быть реализованы на базе разных внутренних структур. В главе 4 была представлена реализация приоритетной очереди на базе упорядоченного массива. У такого решения имеется недостаток: хотя извлечение наибольшего элемента выполняется очень быстро, за время $O(1)$, вставка выполняется медленно, за время $O(N)$, потому что вставка нового элемента в порядке сортировки требует сдвига в среднем половины элементов массива.

В этой главе описана другая структура, которая может использоваться для реализации приоритетной очереди: *пирамида* (heap). Пирамида является разновидностью дерева и обеспечивает вставку и удаление за время $O(\log N)$. Иначе говоря, она не так быстро выполняет удаление, но зато значительно ускоряет вставку. Именно эту структуру данных следует применять для реализации приоритетных очередей, когда важна высокая скорость, а операции вставки достаточно многочисленны.

Общие сведения

Пирамидой называется двоичное дерево, обладающее следующими характеристиками:

- ✦ Полнота. Все уровни дерева содержат все возможные узлы (хотя последний уровень может быть заполнен лишь частично). На рис. 12.1 приведены примеры полных и неполных деревьев.
- ✦ Пирамида (обычно) реализуется на базе массива. Реализация двоичных деревьев в виде массивов (в отличие от хранения ссылок, определяющих связи между узлами) была описана в главе 8, «Двоичные деревья».
- ✦ Для каждого узла в пирамиде выполняется основное условие, гласящее, что ключ каждого узла больше (либо равен) ключей его потомков.

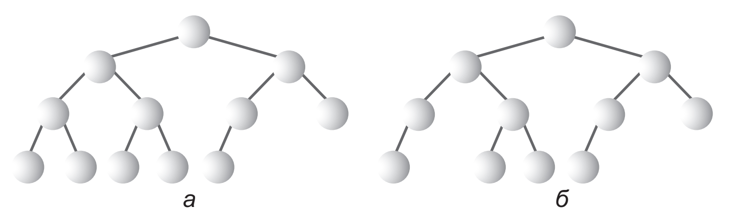


Рис. 12.1. Полные и неполные двоичные деревья: *а* — полное дерево; *б* — неполное дерево

На рис. 12.2 показана пирамида и ее связь с массивом, на базе которого она реализована. В памяти хранится массив, а пирамида является лишь концептуальным представлением. Обратите внимание: дерево является полным, а условие пирамиды выполняется для всех узлов.

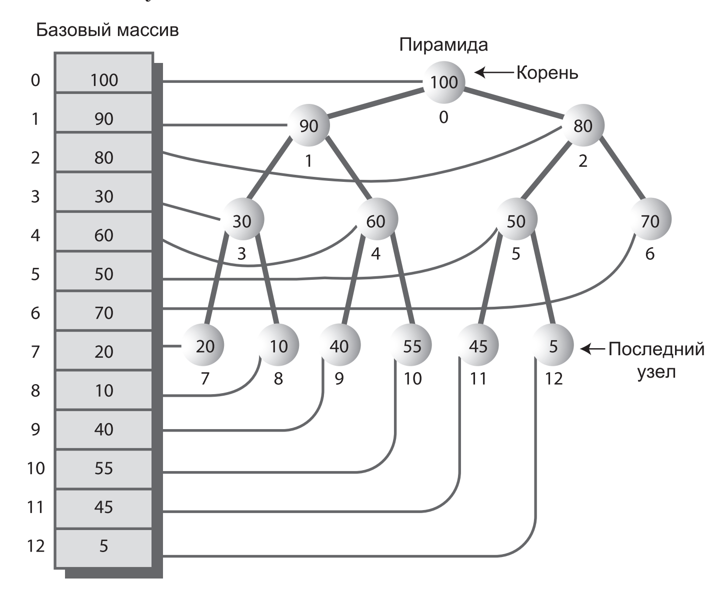


Рис. 12.2. Пирамида и базовый массив

Из того факта, что пирамида является полным двоичным деревом, следует, что в массиве, который используется для ее представления, нет «дыр». Заполнены все ячейки, от 0 до $N - 1$ (на рис. 12.2 $N = 13$).

В этой главе предполагается, что корневой узел содержит наибольший ключ (вместо наименьшего). Приоритетная очередь, созданная на базе такой пирамиды, упорядочена по убыванию приоритетов. (Очереди, упорядоченные по возрастанию приоритетов, рассматривались в главе 4.)

Приоритетные очереди, пирамиды и ADT

Эта глава посвящена пирамидам, хотя пирамиды в основном рассматриваются в контексте реализации приоритетных очередей. Впрочем, между приоритетной очередью и пирамидой, используемой для ее реализации, существует очень тесная связь. Эта связь продемонстрирована в следующем сокращенном фрагменте кода:

```
class Heap
   {
   private Node heapArray[];

   public void insert(Node nd)
      {  }
   public Node remove()
      {  }
   }
class priorityQueue
   {
   private Heap theHeap;

   public void insert(Node nd)
      { theHeap.insert(nd); }
   public Node remove()
      { return theHeap.remove() }
   }
```

Методы класса `priorityQueue` представляют собой простые «обертки» для методов нижележащего класса `Heap`; они обладают той же функциональностью. Пример ясно показывает, что приоритетная очередь представляет собой абстрактный тип данных, который может быть реализован разными способами, тогда как пирамида относится к числу более фундаментальных структур данных. В этой главе для простоты методы пирамиды приводятся в исходном виде, без «упаковки» в методы приоритетной очереди.

Слабая упорядоченность

По сравнению с деревом двоичного поиска, в котором ключ левого потомка каждого узла меньше ключа правого потомка, пирамида является *слабо упорядоченной* (квазиупорядоченной). В частности, ограничения дерева двоичного поиска позволяют выполнить перебор узлов в порядке сортировки по простому алгоритму.

В пирамиде упорядоченный перебор узлов затрудняется тем, что принцип организации пирамиды (условие пирамиды) не так силен, как принцип организации дерева. Единственное, что можно гарантировать по поводу пирамиды — то, что на каждом пути от корня к листу узлы упорядочены по убыванию. Как видно из рис. 12.2, ключи узлов, находящихся слева или справа от заданного узла, на более высоких или низких уровнях (и не принадлежащих тому же пути), могут быть больше или меньше ключа узла. Пути, не имеющие общих узлов, полностью независимы друг от друга.

Вследствие слабой упорядоченности пирамиды некоторые операции с ней затруднены или невозможны. Из-за отсутствия нормальной возможности перебора пирамида не предоставляет удобных средств поиска заданного ключа. Дело в том, что алгоритм поиска не располагает достаточной информацией для принятия решения о том, какого из двух потомков узла следует выбрать для перехода на нижней уровень. Соответственно узел с заданным ключом невозможно удалить (по крайней мере за время $O(\log N)$), потому что его нельзя найти. (Операции можно выполнить последовательным просмотром всех ячеек последовательности, но это может быть сделано только за время $O(N)$.)

Таким образом, структура пирамиды выглядит довольно хаотично. Тем не менее упорядоченности пирамиды достаточно для быстрого удаления наибольшего узла и быстрой вставки новых узлов. Этих операций достаточно для использования пирамиды в качестве приоритетной очереди. Сначала мы в общих чертах разберем, как выполняются эти операции, а затем рассмотрим практические примеры в приложении Workshop.

Удаление

Под удалением подразумевается удаление с наибольшим ключом. Этот узел всегда является корневым, поэтому его удаление выполняется просто. Корень всегда хранится в ячейке 0 базового массива:

```
maxNode = heapArray[0];
```

Проблема в том, что после удаления корня дерево перестает быть полным; в нем появляется пустая ячейка. «Дыру» необходимо заполнить. Все элементы массива можно было бы сдвинуть на одну ячейку, но существует другое, более быстрое решение. Последовательность действий при удалении наибольшего узла выглядит так:

1. Удалить корневой узел.
2. Переместить последний узел на место корневого.
3. Смещать его вниз до тех пор, пока он не окажется ниже большего и выше меньшего узла.

Последним узлом является крайний правый узел самого нижнего занятого уровня дерева. Он соответствует последней заполненной ячейке массива. (На рис. 12.2 это узел с индексом 12 и значением 5.) Копирование этого узла в корень выполняется тривиально:

```
heapArray[0] = heapArray[N-1];
N--;
```

Удаление корня приводит к уменьшению размера массива на 1.

В ходе смещения узел последовательно меняется местами с узлом, находящимся перед ним; после каждой перестановки алгоритм проверяет, оказался ли узел в правильной позиции. На шаге 3 новый корневой узел слишком мал для своей позиции, поэтому он смещается вниз на свое законное место. Код выполнения этой операции будет приведен ниже.

Шаг 2 восстанавливает полноту пирамиды (отсутствие пустых ячеек), а шаг 3 восстанавливает условие пирамиды (каждый узел больше своих потомков). Процесс удаления показан на рис. 12.3.

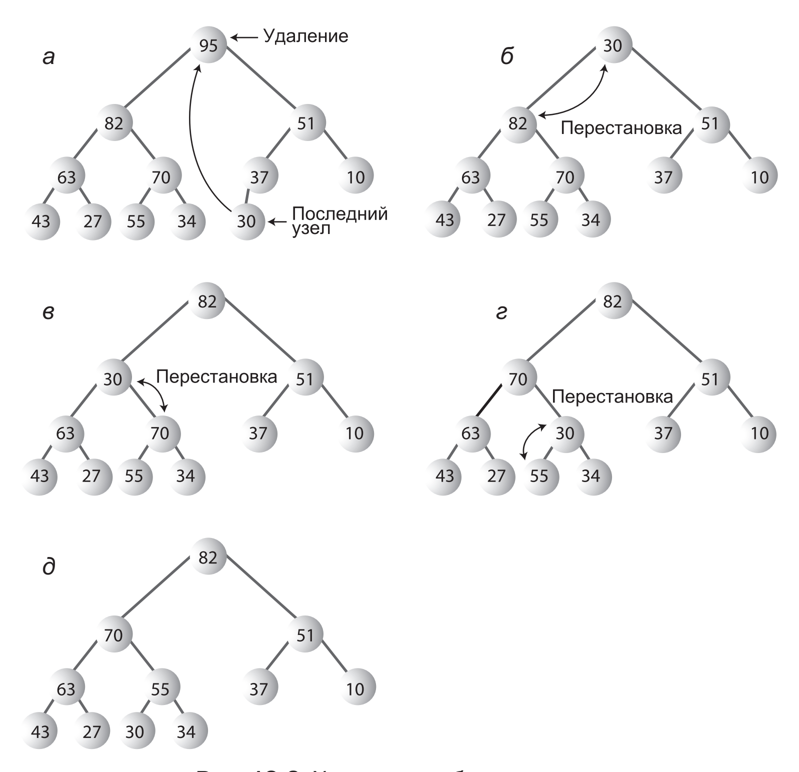


Рис. 12.3. Удаление наибольшего узла

На рис. 12.3, *а* последний узел (30) копируется на место корневого. На рис. 12.3, *б*, *в*, *г* последний узел смещается до своей позиции, которая находится на последнем уровне. (Так бывает не всегда; процесс смещения может остановиться и на одном из промежуточных уровней.) На рис. 12.3, *д* узел занимает правильную позицию.

В каждой промежуточной позиции алгоритм смещения проверяет, какой из потомков больше, и меняет местами целевой узел с бóльшим потомком. Если бы целевой узел поменялся местами с меньшим потомком, то этот потомок стал бы родителем большего потомка, а это было бы нарушением условия пирамиды. Правильные и неправильные перестановки изображены на рис. 12.4.

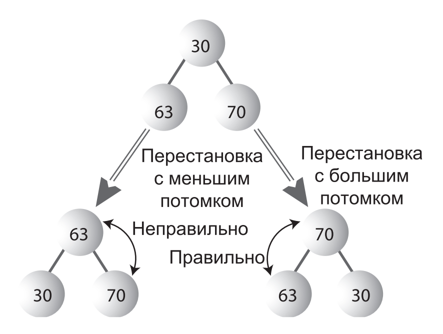


Рис. 12.4. Выбор потомка

Вставка

Вставка узла тоже выполняется относительно просто. При вставке используется смещение вверх (вместо смещения вниз). Сначала вставляемый узел помещается в первую свободную позицию в конце массива, в результате чего размер массива увеличивается на единицу:

```
heapArray[N] = newNode;
N++;
```

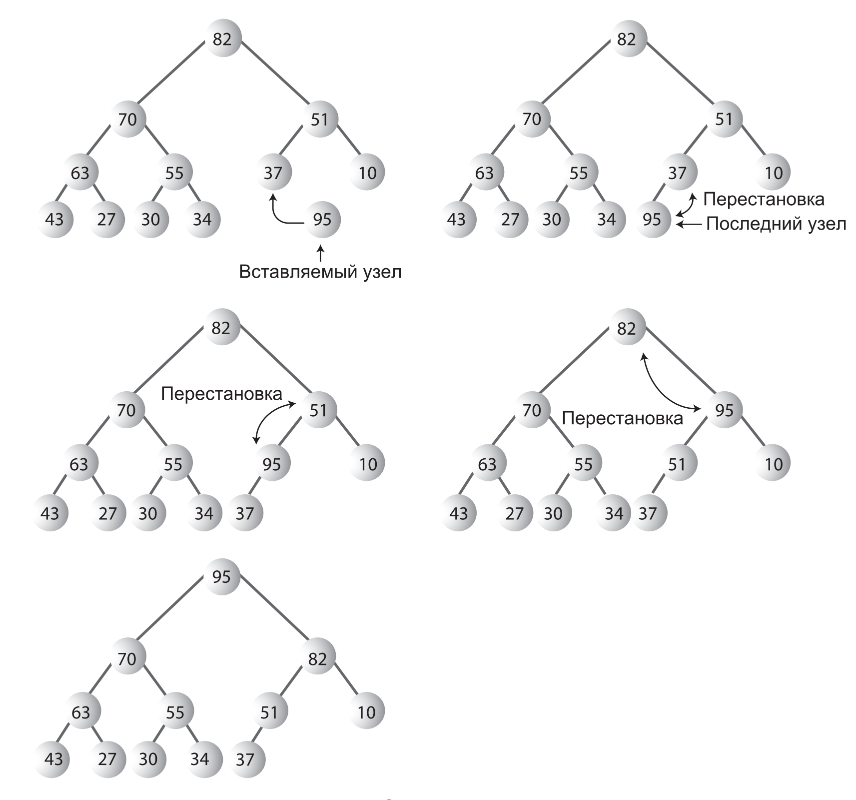


Рис. 12.5. Вставка узла

Такое размещение нарушит условие пирамиды, если ключ нового узла окажется больше ключа его родителя. Так как родитель находится в нижней части пирамиды, его ключ относительно мал, поэтому ключ нового узла с большой вероятностью превысит его. Таким образом, новый узел обычно приходится смещать вверх, пока он не окажется ниже узла с бо́льшим ключом, но выше узла с меньшим ключом. Процесс вставки продемонстрирован на рис. 12.5.

Алгоритм смещения вверх проще алгоритма смещения вниз, потому что он не требует сравнения двух потомков. Узел имеет только одного родителя, так что вставляемый узел просто меняется местами с родителем. На рисунке правильной позицией нового узла оказывается корень дерева, но смещение также может остановиться на одном из промежуточных уровней.

Сравнивая рис. 12.4 и 12.5, легко убедиться, что удаление с последующей вставкой того же узла не обязательно приводит к восстановлению исходной пирамиды. Заданный набор узлов может иметь много действительных представлений в виде пирамиды в зависимости от порядка вставки узлов.

Условные перестановки

На рис. 12.4 и 12.5 показано, как узлы меняются местами в процессе смещения вверх и вниз. На концептуальном уровне такие перестановки упрощают понимание вставок и удалений, и в некоторых реализациях узлы действительно меняются местами. На рис. 12.6, *а* показана упрощенная версия перестановок в процессе смещения вниз. После трех перестановок узел A оказывается в позиции D, а каждый из узлов B, C и D поднимается на один уровень.

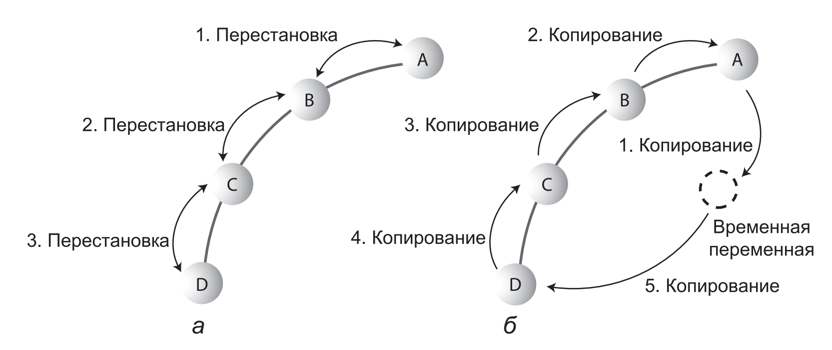


Рис. 12.6. Смещение с перестановками и копированием: *а* — перестановки; *б* — копирование

Однако перестановка требует трех операций копирования; соответственно для трех перестановок на рис. 12.6, *а* потребуется девять копирований. Чтобы сократить общее количество операций копирования при смещении узла, можно заменить перестановки копированиями.

На рис. 12.6, *б* показано, как пять операций копирования выполняют работу трех перестановок. Сначала узел A сохраняется во временной переменной. Затем узел B копируется на место A, узел C — на место B, а узел D — на место C. Наконец,

узел A возвращается из временной переменной на место D. Количество операций копирования сократилось с 9 до 5.

На рисунке узел A перемещается на три уровня. Экономия времени значительно возрастает с увеличением количества уровней, так как две операции копирования во временную переменную и обратно заменяют большее количество операций. Для дерева с многими уровнями сложность сокращается примерно втрое.

Процессы смещения вверх и вниз, выполняемые посредством копирования, также можно представить себе в виде «дыры» (то есть отсутствия узла), движущейся вниз при смещении вверх, и наоборот. Например, на рис. 12.6, *б* копирование A во временную переменную создает «дыру» в A. Вообще говоря, ячейка с «дырой» не совсем пустая — в ней продолжает храниться более ранняя копия узла, но для нас это несущественно. При копировании B в A «дыра» перемещается из A в B в направлении, противоположном смещению узла. Шаг за шагом «дыра» постепенно смещается вниз.

Приложение Heap Workshop

Приложение Heap Workshop демонстрирует, как выполняются операции, описанные в предыдущем разделе: поддерживается вставка новых элементов в пирамиду и удаление наибольшего элемента. Кроме того, вы можете изменить приоритет заданного элемента.

Примерный вид рабочей области приложения Heap Workshop при запуске показан на рис. 12.7.

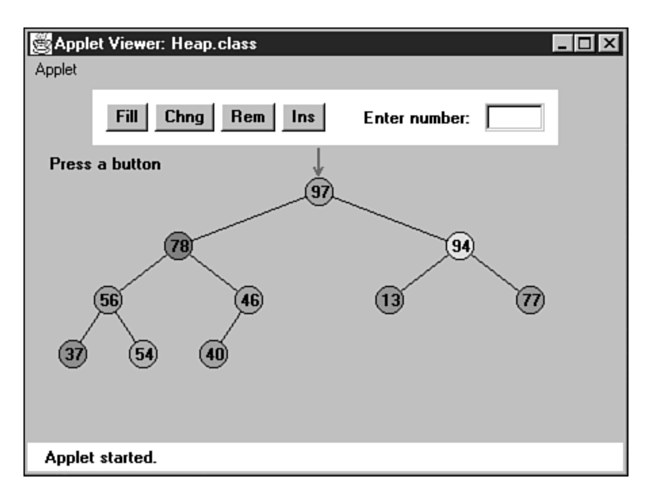


Рис. 12.7. Приложение Heap Workshop

Четыре кнопки — `Fill`, `Chng`, `Rem` и `Ins` — выполняют операции заполнения, изменения приоритета, удаления и вставки соответственно. Давайте посмотрим, как они работают.

Заполнение

При запуске приложения создается пирамида из 10 узлов. При помощи кнопки `Fill` можно создать новую пирамиду с произвольным количеством узлов от 1 до 31. Продолжайте нажимать кнопку `Fill` и введите нужное число по запросу приложения.

Изменение приоритета

Иногда при работе с пирамидой требуется изменить приоритет существующего узла. Такая возможность может пригодиться во многих ситуациях. Скажем, в упомянутом ранее примере с крейсером приближающийся самолет может изменить курс; его приоритет следует понизить в соответствии с изменением обстановки, хотя самолет должен оставаться в очереди до тех пор, пока он не выйдет из зоны действия радара.

Чтобы изменить приоритет узла, нажимайте кнопку `Chng`. При появлении соответствующего запроса щелкните на узле; красная стрелка перемещается к этому узлу. Затем по запросу приложения введите новый приоритет узла.

Если приоритет узла повышается, он смещается вверх к новой позиции. В случае снижения приоритета узел смещается вниз.

Удаление

Кнопка `Rem` удаляет узел с наибольшим ключом, находящийся в корневом узле дерева. Вы увидите, как узел исчезает, а на его месте появляется последний (крайний правый) узел нижнего уровня. Перемещенный узел смещается вниз до тех пор, пока не займет позицию, восстанавливающую иерархию пирамиды.

Вставка

Новый узел всегда изначально вставляется в первую свободную ячейку массива, справа от последнего узла на нижнем уровне пирамиды. Вставленный узел смещается вверх до подходящей позиции. Эта операция выполняется многократным нажатием кнопки `Ins`.

Реализация пирамиды на языке Java

Полный код программы `heap.java` приводится позднее в этом разделе. Но прежде чем рассматривать его, стоит подробнее остановиться на отдельных операциях вставки, удаления и изменения приоритета.

Вспомним несколько фактов из главы 8, касающихся представления дерева в виде массива. Для узла с индексом `x` в массиве:

- индекс родителя равен (`x` – 1)/2;
- индекс левого потомка равен 2*`x` + 1;
- индекс правого потомка равен 2*`x` + 2.

Эти отношения показаны на рис. 12.2.

ПРИМЕЧАНИЕ

Для целых чисел символом / обозначается целочисленное деление с округлением результата в меньшую сторону.

Вставка

Алгоритм смещения вверх выделяется в отдельный метод `trickleUp()`. Код метода `insert()`, включающий вызов `trickleUp()`, получается очень простым:

```
public boolean insert(int key)
   {
   if(currentSize==maxSize)            // Если массив заполнен,
      return false;                    // возвращается признак неудачи
   Node newNode = new Node(key);       // Создание нового узла
   heapArray[currentSize] = newNode;   // Размещение в конце массива
   trickleUp(currentSize++);           // Смещение вверх
   return true;                        // Признак успешной вставки
   }
```

Сначала алгоритм проверяет, что в массиве осталось свободное место, после чего создает новый узел с ключом, переданным в аргументе. Узел вставляется в конец массива. Наконец, вызов метода `trickleUp()` перемещает узел в правильную позицию.

В аргументе метода `trickleUp()` (см. ниже) передается индекс вставленного элемента. Алгоритм находит родителя узла, находящегося в этой позиции, и сохраняет узел в переменной `bottom`. В цикле `while` переменная `index` смещается вверх по пути к корневому узлу, последовательно указывая на каждый узел. Цикл `while` выполняется, пока не был достигнут корневой узел (`index>0`), а ключ (`iData`) родителя `index` меньше ключа нового узла.

В теле цикла `while` выполняется один шаг процесса смещения вверх. Сначала родительский узел копируется в ячейку `index` («дыра» поднимается вверх). Затем `index` перемещается вверх присваиванием индекса родительского узла, а в индекс родительского узла заносится индекс *его* родителя.

```
public void trickleUp(int index)
   {
   int parent = (index-1) / 2;
   Node bottom = heapArray[index];

   while( index > 0 &&
          heapArray[parent].getKey() < bottom.getKey() )
      {
      heapArray[index] = heapArray[parent]; // Узел перемещается вниз
      index = parent;                       // index перемещается вверх
      parent = (parent-1) / 2;              // parent <- его родитель
      }
   heapArray[index] = bottom;
   }
```

При выходе из цикла только что вставленный узел, хранившийся в `bottom`, вставляется в ячейку, на которую ссылается `index`. Это первая ячейка, в которой узел окажется не больше своего родителя, поэтому вставка в нее выполняет условие пирамиды.

Удаление

Алгоритм удаления тоже получается несложным, если выделить алгоритм смещения вниз в отдельную функцию. Мы сохраняем узел из корня, копируем последний узел (с индексом currentSize-1) на место корня и вызываем trickleDown() для перемещения этого узла в подходящее место.

```
public Node remove()           // Удаление элемента с максимальным ключом
   {                           // (Предполагается, что список не пуст)
   Node root = heapArray[0];   // Сохранение корня
   heapArray[0] = heapArray[--currentSize];  // Корень <- последний узел
   trickleDown(0);             // Корневой узел смещается вниз
   return root;                // Метод возвращает удаленный узел
   }
```

Метод возвращает удаленный узел, который обычно каким-то образом обрабатывается пользователем пирамиды.

Метод смещения вниз trickleDown() сложнее метода trickleUp(), потому что мы должны определить, какой из двух потомков больше. Сначала узел с индексом index сохраняется в переменной с именем top. Если метод trickleDown() был вызван из remove(), то index ссылается на корневой узел, но как будет показано ниже, он также может вызываться из других методов.

Цикл while выполняется до тех пор, пока index не окажется на нижнем уровне дерева, то есть пока у узла имеется хотя бы один потомок. В цикле алгоритм проверяет, существует ли у узла правый потомок (некоторые узлы могут иметь только левого потомка), и если существует — сравнивает ключи потомков, присваивая largerChild индекс большего потомка.

Затем алгоритм проверяет, что ключ исходного узла (который теперь находится в top) больше ключа largerChild. Если условие выполнено, то процесс смещения вниз завершен, и выполнение цикла прерывается.

```
public void trickleDown(int index)
   {
   int largerChild;
   Node top = heapArray[index];       // Сохранение корня
   while(index < currentSize/2)       // Пока у узла имеется
      {                               // хотя бы один потомок
      int leftChild = 2*index+1;
      int rightChild = leftChild+1;
                                      // Определение большего потомка
      if( rightChild < currentSize &&  // (Правый потомок существует?)
                    heapArray[leftChild].getKey() <
                    heapArray[rightChild].getKey() )
         largerChild = rightChild;
      else
         largerChild = leftChild;
                                       // top >= largerChild?
      if(top.getKey() >= heapArray[largerChild].getKey())
```

```
         break;
                                     // Потомок сдвигается вверх
      heapArray[index] = heapArray[largerChild];
      index = largerChild;          // Переход вниз
      }
   heapArray[index] = top;          // index <- корень
   }
```

При выходе из цикла остается лишь вернуть узел, хранящийся в top, на его законное место, на которое указывает переменная index.

Изменение ключа

С готовыми методами trickleDown() и trickleUp() мы можем легко реализовать алгоритм изменения приоритета (ключа) узла, с последующим смещением узла вверх или вниз до нужной позиции. Операция изменения ключа выполняется методом change():

```
public boolean change(int index, int newValue)
   {
   if(index<0 || index>=currentSize)
      return false;
   int oldValue = heapArray[index].getKey(); // Сохранение старого ключа
   heapArray[index].setKey(newValue);  // Присваивание нового ключа

   if(oldValue < newValue)             // Если узел повышается,
      trickleUp(index);                // выполняется смещение вверх
   else                                // Если узел понижается,
      trickleDown(index);              // выполняется смещение вниз
   return true;
   }
```

Метод сначала проверяет, что в первом аргументе был передан действительный индекс, и если проверка проходит успешно — записывает в поле iData узла с заданным индексом значение, переданное во втором аргументе. Если приоритет увеличился, то узел смещается вверх, а если уменьшился — смещается вниз.

Вообще говоря, в методе не показана самая трудная часть изменения приоритета: поиск изменяемого узла. В приведенном методе change() индекс передается в аргументе, а в приложении Heap Workshop пользователь просто щелкает на нужном узле. В реальном приложении потребуется механизм поиска правильного узла; как было сказано ранее, в пирамиде можно легко и просто обратиться только к одному узлу — корневому с наибольшим ключом.

Задача может быть решена последовательным поиском по массиву с линейной сложностью $O(N)$. Также каждое появление узла в приоритетной очереди может сопровождаться обновлением другой структуры данных (например, хеш-таблицы). Отдельная структура данных обеспечит быстрый доступ к любому узлу, но ее обновление само по себе требует затрат времени.

Размер массива

Размер массива (количество узлов в пирамиде) — важнейший атрибут состояния пирамиды и необходимое поле класса Heap. Узлы, скопированные из последней ячейки, не стираются, поэтому алгоритм может определить положение последней занятой ячейки только по текущему размеру массива.

Программа heap.java

В программе heap.java (листинг 12.1) используется класс Node. Единственным полем этого класса является переменная iData, в которой хранится ключ узла. Естественно, в реальной программе этот класс содержал бы много других полей. Класс Heap содержит методы, упомянутые выше, а также методы isEmpty() и displayHeap(). Последний метод выводит примитивное, но наглядное символьное представление содержимого пирамиды.

Листинг 12.1. Программа heap.java

```
// heap.java
// Работа с пирамидой
// Запуск программы: C>java HeapApp
import java.io.*;
////////////////////////////////////////////////////////////////
class Node
   {
   private int iData;             // Данные (ключ)
// -------------------------------------------------------------
   public Node(int key)           // Конструктор
      { iData = key; }
// -------------------------------------------------------------
   public int getKey()
      { return iData; }
// -------------------------------------------------------------
   public void setKey(int id)
      { iData = id; }
// -------------------------------------------------------------
   }  // Конец класса Node
////////////////////////////////////////////////////////////////
class Heap
   {
   private Node[] heapArray;
   private int maxSize;           // Размер массива
   private int currentSize;       // Количество узлов в массиве
// -------------------------------------------------------------
   public Heap(int mx)            // Конструктор
      {
      maxSize = mx;
      currentSize = 0;
```

```
      heapArray = new Node[maxSize];  // Создание массива
      }
// -------------------------------------------------------------
   public boolean isEmpty()
      { return currentSize==0; }
// -------------------------------------------------------------
   public boolean insert(int key)
      {
      if(currentSize==maxSize)
         return false;
      Node newNode = new Node(key);
      heapArray[currentSize] = newNode;
      trickleUp(currentSize++);
      return true;
   }
// -------------------------------------------------------------
   public void trickleUp(int index)
      {
      int parent = (index-1) / 2;
      Node bottom = heapArray[index];
      while( index > 0 &&
            heapArray[parent].getKey() < bottom.getKey() )
         {
         heapArray[index] = heapArray[parent];  // Узел смещается вниз
         index = parent;
         parent = (parent-1) / 2;
      }
      heapArray[index] = bottom;
   }
// -------------------------------------------------------------
   public Node remove()           // Удаление элемента с наибольшим ключом
      {                           // (Предполагается, что список не пуст)
      Node root = heapArray[0];
      heapArray[0] = heapArray[--currentSize];
      trickleDown(0);
      return root;
   }
// -------------------------------------------------------------
   public void trickleDown(int index)
      {
      int largerChild;
      Node top = heapArray[index];       // Сохранение корня
      while(index < currentSize/2)       // Пока у узла имеется
         {                               // хотя бы один потомок
         int leftChild = 2*index+1;
         int rightChild = leftChild+1;
                                         // Определение большего потомка
         if(rightChild < currentSize &&  // (Правый потомок существует?)
                     heapArray[leftChild].getKey() <
```

продолжение ⇨

Листинг 12.1 (*продолжение*)

```
                              heapArray[rightChild].getKey())
            largerChild = rightChild;
         else
            largerChild = leftChild;
                                        // top >= largerChild?
         if( top.getKey() >= heapArray[largerChild].getKey() )
            break;
                                        // Потомок сдвигается вверх
         heapArray[index] = heapArray[largerChild];
         index = largerChild;           // Переход вниз
         }
      heapArray[index] = top;           // index <- корень
   }
// -------------------------------------------------------------
   public boolean change(int index, int newValue)
      {
      if(index<0 || index>=currentSize)
         return false;
      int oldValue = heapArray[index].getKey(); // Сохранение старого ключа
      heapArray[index].setKey(newValue);  // Присваивание нового ключа

      if(oldValue < newValue)             // Если узел повышается,
         trickleUp(index);                // выполняется смещение вверх.
      else                                // Если узел понижается,
         trickleDown(index);              // выполняется смещение вниз.
      return true;
      }
// -------------------------------------------------------------
   public void displayHeap()
      {
      System.out.print("heapArray: ");    // Формат массива
      for(int m=0; m<currentSize; m++)
         if(heapArray[m] != null)
            System.out.print( heapArray[m].getKey() + " ");
         else
            System.out.print( "-- ");
      System.out.println();
                                          // Формат пирамиды
      int nBlanks = 32;
      int itemsPerRow = 1;
      int column = 0;
      int j = 0;                          // Текущий элемент
      String dots = "...............................";
      System.out.println(dots+dots);      // Верхний пунктир

      while(currentSize > 0)              // Для каждого элемента пирамиды
         {
         if(column == 0)                  // Первый элемент в строке?
            for(int k=0; k<nBlanks; k++)  // Предшествующие пробелы
```

```
            System.out.print(' ');
                                      // Вывод элемента
         System.out.print(heapArray[j].getKey());
         if(++j == currentSize)           // Вывод завершен?
            break;

         if(++column==itemsPerRow)        // Конец строки?
            {
            nBlanks /= 2;                 // Половина пробелов
            itemsPerRow *= 2;             // Вдвое больше элементов
            column = 0;                   // Начать заново
            System.out.println();         // Переход на новую строку
            }
         else                             // Следующий элемент в строке
            for(int k=0; k<nBlanks*2-2; k++)
               System.out.print(' ');     // Внутренние пробелы
         }
      System.out.println("\n"+dots+dots); // Нижний пунктир
      }
// -------------------------------------------------------------
   }  // Конец класса Heap
////////////////////////////////////////////////////////////////
class HeapApp
   {
   public static void main(String[] args) throws IOException
      {
      int value, value2;
      Heap theHeap = new Heap(31);  // Создание пирамиды с максимальным
      boolean success;              // размером 31

      theHeap.insert(70);           // Вставка 10 items
      theHeap.insert(40);
      theHeap.insert(50);
      theHeap.insert(20);
      theHeap.insert(60);
      theHeap.insert(100);
      theHeap.insert(80);
      theHeap.insert(30);
      theHeap.insert(10);
      theHeap.insert(90);

      while(true)           // Пока пользователь не нажмет Ctrl+C
         {
         System.out.print("Enter first letter of ");
         System.out.print("show, insert, remove, change: ");
         int choice = getChar();
         switch(choice)
            {
            case 's':                       // Вывод
```

продолжение ⇨

Листинг 12.1 (*продолжение*)

```
            theHeap.displayHeap();
            break;
         case 'i':                        // Вставка
            System.out.print("Enter value to insert: ");
            value = getInt();
            success = theHeap.insert(value);
            if( !success )
               System.out.println("Can't insert; heap full");
            break;
         case 'r':                        // Удаление
            if( !theHeap.isEmpty() )
               theHeap.remove();
            else
               System.out.println("Can't remove; heap empty");
            break;
         case 'c':                        // Изменение приоритета
            System.out.print("Enter current index of item: ");
            value = getInt();
            System.out.print("Enter new key: ");
            value2 = getInt();
            success = theHeap.change(value, value2);
            if( !success )
               System.out.println("Invalid index");
            break;
         default:
            System.out.println("Invalid entry\n");
         }
      }
   }
//-------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
}
//-------------------------------------------------------------
   public static char getChar() throws IOException
      {
      String s = getString();
      return s.charAt(0);
      }
//-------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
      return Integer.parseInt(s);
```

```
      }
//-------------------------------------------------------------
   }  // Конец класса HeapApp
///////////////////////////////////////////////////////////////
```

В массиве корень пирамиды размещается в ячейке с индексом 0. Некоторые реализации пирамиды размещают корень в ячейке 1, а ячейка 0 используется в качестве «сторожа» с наибольшим возможным ключом. Такое решение немного ускоряет некоторые алгоритмы, но усложняет реализацию на концептуальном уровне.

Метод main() приложения HeapApp создает пирамиду с максимальным размером 31 (обусловленным ограничениями метода вывода) и вставляет в нее 10 узлов со случайными ключами. Затем программа входит в цикл, в котором пользователь вводит команды *s*, *i*, *r* или *c* (соответственно вывод, вставка, удаление или изменение).

Пример взаимодействия с программой:

```
Enter first letter of show, insert, remove, change: s
heapArray: 100 90 80 30 60 50 70 20 10 40
..............................................................
                                100
                90                              80
        30              60              50              70
    20      10      40
..............................................................
Enter first letter of show, insert, remove, change: i
Enter value to insert: 53
Enter first letter of show, insert, remove, change: s
heapArray: 100 90 80 30 60 50 70 20 10 40 53
..............................................................
                                100
                90                              80
        30              60              50              70
    20      10      40      53
..............................................................
Enter first letter of show, insert, remove, change: r
Enter first letter of show, insert, remove, change: s
heapArray: 90 60 80 30 53 50 70 20 10 40
..............................................................
                                 90
                60                              80
        30              53              50              70
    20      10      40
..............................................................
Enter first letter of show, insert, remove, change:
```

Пользователь выводит состояние пирамиды, добавляет элемент с ключом 53, снова выводит, удаляет элемент с наибольшим ключом и выводит состояние пирамиды в третий раз. Метод show() выводит оба представления пирамиды: как в виде массива, так и в виде дерева. Сила воображения поможет вам заполнить связи между узлами.

Расширение массива

Что произойдет, если во время выполнения программы в массив пирамиды будет вставлено слишком много элементов? Программа может создать новый массив и скопировать в него данные из старого массива. (В отличие от хеш-таблиц изменение размера пирамиды не требует изменения порядка данных.) Операция копирования выполняется за линейное время, но расширение массива выполняется относительно редко, особенно если размер массива существенно увеличивается при каждом расширении (скажем, вдвое).

ПОЛЕЗНЫЙ СОВЕТ

В языке Java вместо массива можно воспользоваться объектом класса Vector — векторы могут расширяться динамически.

Эффективность операций с пирамидой

Для пирамиды со сколько-нибудь значительным количеством элементов алгоритмы смещения вверх и вниз оказываются наиболее затратными составляющими всех представленных операций. Эти алгоритмы выполняются в цикле: узлы шаг за шагом смещаются вверх и вниз по пути. Количество необходимых операций копирования ограничивается высотой пирамиды: если пирамида состоит из пяти уровней, то четыре операции копирования переместят «дыру» от верхнего уровня до нижнего. (Не будем учитывать два перемещения последнего узла во временное хранилище и обратно; они необходимы всегда и поэтому выполняются за постоянное время.)

В цикле метода `trickleUp()` выполняется всего одна существенная операция: сравнение ключа нового узла с ключом узла в текущей позиции. Метод `trickleDown()` требует двух сравнений: для поиска большего потомка и для его сравнения с «последним» узлом. В обоих случаях операция завершается копированием узла сверху вниз или снизу вверх.

Пирамида является частным случаем двоичного дерева, а как было показано в главе 8, количество уровней L в двоичном дереве равно $\log_2(N + 1)$, где N — количество узлов. Циклы методов `trickleUp()` и `trickleDown()` выполняются за $L - 1$ итераций, поэтому время выполнения первого пропорционально $\log_2 N$, а время выполнения второго чуть больше из-за дополнительного сравнения. Таким образом, все операции с пирамидой, рассмотренные в этом разделе, выполняются за время $O(\log N)$.

Пирамидальное дерево

На иллюстрациях этой главы пирамиды изображались в виде деревьев, потому что такое представление получается более наглядным. При этом для реализации пирамиды был выбран массив. Тем не менее пирамида может быть реализована и на базе дерева. Такое дерево будет двоичным, но оно не будет деревом двоичного

поиска, потому что, как мы уже видели, оно обладает слабой упорядоченностью. Кроме того, дерево будет полным (то есть в нем не будет отсутствующих узлов). Мы будем называть такое дерево *пирамидальным*.

Одна из проблем при работе с пирамидальными деревьями связана с поиском последнего узла. Эта операция необходима для удаления наибольшего элемента, потому что это последний узел, который вставляется на место удаляемого корня (а затем смещается вниз). Также может возникнуть необходимость в поиске первого пустого узла, потому что в этом месте вставляется новый узел (с последующим смещением вверх). Обнаружить эти узлы посредством поиска не удастся, потому что их значения неизвестны, к тому же дерево не является деревом поиска. Но если реализация отслеживает количество узлов, найти их в полном дереве будет несложно.

Как было показано при обсуждении дерева Хаффмана в главе 8, путь от корня до листа может быть представлен в виде двоичного числа. Двоичные цифры этого числа обозначают направление перехода от каждого родителя к потомку: 0 для левого, 1 для правого.

Оказывается, между количеством узлов в дереве и двоичным числом, кодирующим путь к последнему узлу, существует простая связь. Допустим, корню присвоен номер 1; на втором уровне находятся узлы 2 и 3; на третьем — узлы 4, 5, 6, 7 и т. д. Возьмите номер искомого узла — последнего узла или первого пустого узла. Преобразуйте номер узла в двоичное представление. Предположим, дерево состоит из 29 узлов, и вы хотите найти последний узел. Десятичное число 29 соответствует 11101 в двоичной записи. Удалите начальную единицу; остается 1101. Эта запись определяет путь от корня к узлу 29: направо, направо, налево, направо. Номер первого свободного узла 30 в двоичной записи имеет вид 1110 (после удаления начальной единицы): направо, направо, направо, налево.

Двоичное представление числа может быть получено многократным применением оператора `%` для получения остатка от деления номера узла на 2 (0 или 1) и последующего целочисленного деления n на 2 оператором `/`. Когда значение n станет меньше единицы, вычисления закончены. Последовательность остатков, которую можно сохранить в массиве или строке, определяет двоичное число. (Младшие биты находятся в начале строки.) Реализация может выглядеть так:

```
while(n >= 1)
   {
   path[j++] = n % 2;
   n = n / 2;
   }
```

Также существует рекурсивное решение: остатки вычисляются при каждом рекурсивном вызове, а соответствующее направление выбирается при возврате управления.

После того как нужный узел (или пустой потомок) будет найден, операции с пирамидой становятся тривиальными. При смещении вверх или вниз структура дерева не изменяется, поэтому выполнять фактическое перемещение узлов не нужно — достаточно копировать данные из одного узла в следующий. Это позволяет избежать присоединения и отсоединения всех потомков и родителей при простом

перемещении. Класс Node должен содержать поле родительского узла, потому что смещение вверх требует обращения к родителю. Реализация пирамидального дерева предоставляется читателю в виде программного проекта.

Операции с пирамидальным деревом выполняются за время $O(\log N)$. Как и при реализации пирамиды на базе массива, время тратится в основном на операции смещения вверх и вниз, сложность которых пропорциональна высоте дерева.

Пирамидальная сортировка

Эффективность пирамиды как структуры данных является одной из предпосылок на удивление простого и эффективного алгоритма сортировки, называемого *пирамидальной сортировкой*.

Алгоритм строится на вставке всех неупорядоченных элементов в пирамиду обычным методом insert(). Затем многократные вызовы remove() извлекают элементы в порядке сортировки. Реализация может выглядеть примерно так:

```
for(j=0; j<size; j++)
   theHeap.insert( anArray[j] );   // Из несортированного массива
for(j=0; j<size; j++)
   anArray[j] = theHeap.remove();  // В отсортированный массив
```

Так как методы insert() и remove() выполняются за время $O(\log N)$, и каждый из них должен быть выполнена N раз, вся сортировка выполняется за время $O(N*\log N)$ — такое же, как при быстрой сортировке. Однако по скорости этот алгоритм все же уступает быстрой сортировке — отчасти из-за того, что во внутреннем цикле while метода trickleDown() выполняется больше операций, чем во внутреннем цикле быстрой сортировки.

Однако существует пара приемов, повышающих эффективность пирамидальной сортировки. Первый экономит время, а второй — память.

Ускоренное смещение вниз

Вставляя в пирамиду N новых элементов, мы применяем метод trickleUp() N раз. Однако все элементы можно вставить в случайных позициях массива, а затем восстановить иерархию пирамиды всего $N/2$ применениями trickleDown(). Этот прием обеспечивает небольшой прирост скорости.

Две правильные субпирамиды образуют правильную пирамиду

Чтобы понять, как работает это решение, необходимо знать, что trickleDown() создает правильную пирамиду в том случае, если при нарушении корневым элементом условия пирамиды обе дочерних субпирамиды этого корня являются правильными пирамидами. (Корневой элемент может быть корнем как субпирамиды, так и всей пирамиды.) Ситуация показана на рис. 12.8.

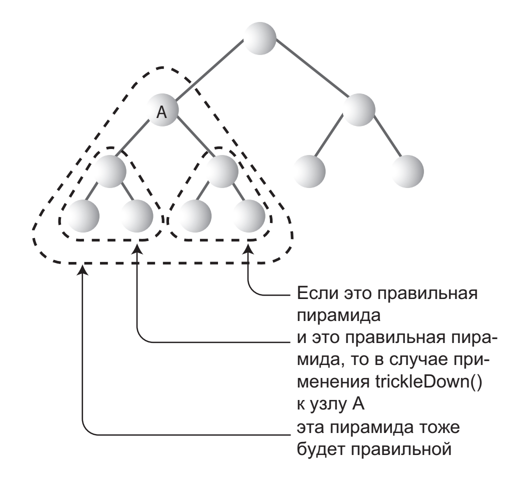


Рис. 12.8. Обе субпирамиды должны быть правильными

Из этого факта следует способ преобразования неупорядоченного массива в пирамиду. Мы можем применить `trickleDown()` к узлам у низа (будущей) пирамиды, то есть в конце массива, и двигаться вверх к корню с индексом 0. На каждом шаге нижние субпирамиды будут правильными, потому что метод `trickleDown()` к ним уже был применен. После применения `trickleDown()` к корню неупорядоченный массив будет преобразован в пирамиду.

Однако следует заметить, что узлы нижнего уровня (не имеющие потомков) уже являются правильными пирамидами, потому что они представляют собой деревья из одного узла; у них просто нет связей, которые могли бы быть нарушены. Следовательно, применять к этим узлам `trickleDown()` не нужно. Начинать следует с узла $N/2 - 1$, крайнего правого узла с потомком, вместо последнего узла $N - 1$. А следовательно, по сравнению с N-кратным использованием `insert()` количество смещений сокращается вдвое. На рис. 12.9 показан порядок применения алгоритма смещения вниз, начиная с узла 6 в пирамиде из 15 узлов.

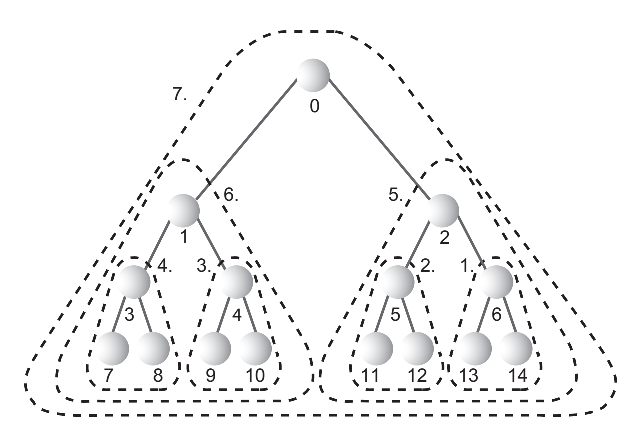


Рис. 12.9. Порядок применения trickleDown()

В следующем фрагменте кода метод trickleDown() применяется ко всем узлам, кроме узлов нижнего уровня, от позиции $N/2 - 1$ и до корня дерева:

```
for(j=size/2-1; j >=0; j--)
   theHeap.trickleDown(j);
```

Рекурсивное решение

Для построения пирамиды из массива также можно воспользоваться рекурсией. Метод heapify() вызывается для корня дерева. Он вызывает себя для двух потомков корня, затем для двух потомков каждого из этих потомков и т. д. В конечном итоге он доберется до нижнего уровня, где немедленно вернет управление при обнаружении узла, не имеющего потомков.

Вызвав себя для двух дочерних поддеревьев, метод heapify() затем вызывает trickleDown() для корня поддерева. Этот вызов гарантирует, что поддерево представляет собой правильную пирамиду. Затем heapify() возвращает управление, а работа с поддеревом продолжается уровнем выше.

```
heapify(int index)      // Преобразование массива в пирамиду
   {
   if(index > N/2-1)    // Если узел не имеет потомков,
      return;           // возврат управления
   heapify(index*2+2);  // Правое поддерево преобразуется в пирамиду
   heapify(index*2+1);  // Левое поддерево преобразуется в пирамиду
   trickleDown(index);  // Узел смещается вниз
   }
```

Вероятно, по эффективности рекурсивное решение уступает простому циклу.

Сортировка «на месте»

В исходном фрагменте кода неупорядоченные данные хранятся в массиве. Затем эти данные вставляются в пирамиду, затем извлекаются из нее и записываются обратно в массив в порядке сортировки. Для этой процедуры необходимы два массива размера *N*: исходный и массив, используемый для хранения пирамиды.

В действительности для хранения пирамиды и исходных данных достаточно одного массива. Это вдвое сокращает объем памяти, необходимой для пирамидальной сортировки; алгоритм не требует дополнительных затрат памяти за пределами исходного массива.

Мы уже видели, что массив может быть преобразован в пирамиду применением trickleDown() к половине элементов массива. Преобразование неупорядоченного массива в пирамиду осуществляется «на месте»; для выполнения этой операции достаточно одного массива. Таким образом, для первого шага пирамидальной сортировки достаточно одного массива.

Однако ситуация усложняется с повторным применением remove() к пирамиде. Где хранить извлекаемые элементы?

При каждом извлечении элемента из пирамиды в конце массива появляется свободная ячейка; размер пирамиды уменьшается на единицу. Только что из-

влеченный элемент можно сохранить в освободившейся ячейке. С извлечением других элементов массив пирамиды становится все меньше, а массив упорядоченных данных — все больше. Таким образом, при соответствующем планировании упорядоченный массив и массив пирамиды смогут совместно использовать одно пространство (рис. 12.10).

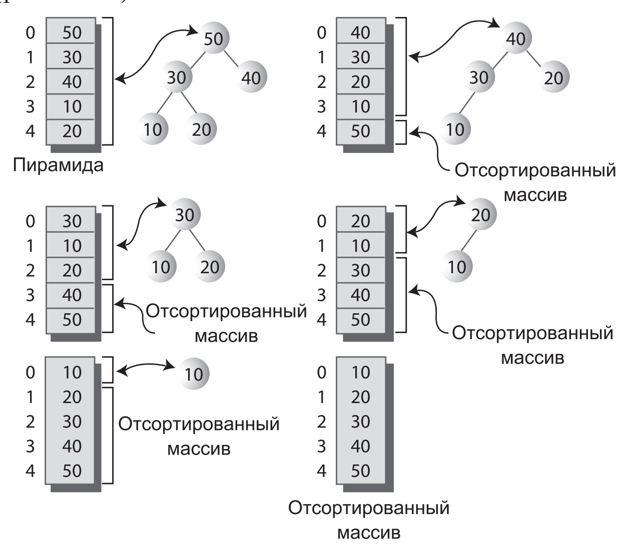


Рис. 12.10. Массив двойного назначения

Программа heapSort.java

Мы объединим эти два приема — применение `trickleDown()` без `insert()` и использование одного массива для хранения исходных данных и пирамиды — в программе для выполнения пирамидальной сортировки. В листинге 12.2 приведен полный код программы `heapSort.java`.

Листинг 12.2. Программа heapSort.java

```
// heapSort.java
// Пирамидальная сортировка
// Запуск программы: C>java HeapSortApp
import java.io.*;
////////////////////////////////////////////////////////////////
class Node
   {
   private int iData;             // Данные (ключ)
// -------------------------------------------------------------
   public Node(int key)           // Конструктор
      { iData = key; }
// -------------------------------------------------------------
```

продолжение ⇨

Листинг 12.2 (*продолжение*)

```
   public int getKey()
      { return iData; }
// -------------------------------------------------------------
   }  // Конец класса Node
////////////////////////////////////////////////////////////////
class Heap
   {
   private Node[] heapArray;
   private int maxSize;           // Размер массива
   private int currentSize;
// -------------------------------------------------------------
   public Heap(int mx)
      {
      maxSize = mx;
      currentSize = 0;
      heapArray = new Node[maxSize];
      }
// -------------------------------------------------------------
   public Node remove()
      {
      Node root = heapArray[0];
      heapArray[0] = heapArray[--currentSize];
      trickleDown(0);
      return root;
      }
// -------------------------------------------------------------
   public void trickleDown(int index)
      {
      int largerChild;
      Node top = heapArray[index];        // Сохранение корня
      while(index < currentSize/2)        // до нижнего уровня
         {
         int leftChild = 2*index+1;
         int rightChild = leftChild+1;
                                          // Определение большего потомка
         if(rightChild < currentSize &&   // (Правый потомок существует?)
                             heapArray[leftChild].getKey() <
                             heapArray[rightChild].getKey())
            largerChild = rightChild;
         else
            largerChild = leftChild;
                                          // top >= largerChild?
         if(top.getKey() >= heapArray[largerChild].getKey())
            break;
                                          // Потомок сдвигается вверх
         heapArray[index] = heapArray[largerChild];
         index = largerChild;             // Переход вниз
         }
      heapArray[index] = top;             // index <- корень
      }
```

```
// -------------------------------------------------------------
   public void displayHeap()
      {
      int nBlanks = 32;
      int itemsPerRow = 1;
      int column = 0;
      int j = 0;                          // Текущий элемент
      String dots = "...............................";
      System.out.println(dots+dots);      // Верхний пунктир

      while(currentSize > 0)              // Для каждого элемента пирамиды
         {
         if(column == 0)                  // Первый элемент в строке?
            for(int k=0; k<nBlanks; k++)  // Предшествующие пробелы
               System.out.print(' ');
                                          // Вывод элемента
         System.out.print(heapArray[j].getKey());

         if(++j == currentSize)           // Вывод завершен?
            break;

         if(++column==itemsPerRow)        // Конец строки?
            {
            nBlanks /= 2;                 // Половина пробелов
            itemsPerRow *= 2;             // Вдвое больше элементов
            column = 0;                   // Начать заново
            System.out.println();         // Переход на новую строку
            }
         else                             // Следующий элемент в строке
            for(int k=0; k<nBlanks*2-2; k++)
               System.out.print(‘ ‘);     // Внутренние пробелы
         }
      System.out.println("\n"+dots+dots); // Нижний пунктир
      }
// -------------------------------------------------------------
   public void displayArray()
      {
      for(int j=0; j<maxSize; j++)
      System.out.print(heapArray[j].getKey() + " ");
      System.out.println("");
      }
// -------------------------------------------------------------
   public void insertAt(int index, Node newNode)
      { heapArray[index] = newNode; }
// -------------------------------------------------------------
   public void incrementSize()
      { currentSize++; }
// -------------------------------------------------------------
   }  // Конец класса Heap
```

продолжение ⇨

Листинг 12.2 (*продолжение*)

```
////////////////////////////////////////////////////////////////
class HeapSortApp
   {
   public static void main(String[] args) throws IOException
      {
      int size, j;
      System.out.print("Enter number of items: ");
      size = getInt();
      Heap theHeap = new Heap(size);

      for(j=0; j<size; j++)       // Заполнение массива
         {                        // случайными данными
         int random = (int)(java.lang.Math.random()*100);
         Node newNode = new Node(random);
         theHeap.insertAt(j, newNode);
         theHeap.incrementSize();
         }

         System.out.print("Random: ");
            theHeap.displayArray();  // Вывод случайного массива

         for(j=size/2-1; j>=0; j--)  // Преобразование массива в пирамиду
            theHeap.trickleDown(j);

         System.out.print("Heap:   ");
         theHeap.displayArray();     // Вывод массива
         theHeap.displayHeap();      // Вывод пирамиды

         for(j=size-1; j>=0; j--)    // Извлечение из пирамиды
            {                        // с сохранением в конце массива
            Node biggestNode = theHeap.remove();
            theHeap.insertAt(j, biggestNode);
            }
         System.out.print("Sorted: ");
         theHeap.displayArray();     // Вывод отсортированного массива
         }
// -------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//-------------------------------------------------------------
   public static int getInt() throws IOException
      {
      String s = getString();
```

```
      return Integer.parseInt(s);
      }
// -------------------------------------------------------------
   }  // Конец класса HeapSortApp
```

Класс Heap почти не отличается от одноименного класса из программы heap.java (см. листинг 12.1), если не считать того, что для экономии места из него были исключены методы trickleUp() и insert(), лишние для пирамидальной сортировки. В класс включен метод insertAt() для прямой вставки в массив пирамиды.

Учтите, что это добавление не соответствует духу объектно-ориентированного программирования. Интерфейс класса Heap должен ограждать пользователей класса от подробностей реализации пирамиды. Базовый массив должен оставаться невидимым, однако метод insertAt() позволяет выполнять прямую вставку. В такой ситуации нарушение принципов ООП приемлемо, потому что массив так тесно связан с архитектурой пирамиды.

В классе пирамиды также появился новый метод incrementSize(). Может показаться, что его можно объединить с insertAt(), но при вставке в массив в его качестве упорядоченного массива размер пирамиды увеличиваться не должен, поэтому мы разделяем эти две функции.

Метод main() класса HeapSortApp:

1) запрашивает у пользователя размер массива;
2) заполняет массив случайными данными;
3) преобразует массив в пирамиду $N/2$ с применениями метода trickleDown();
4) извлекает элементы из кучи и записывает их обратно в конец массива.

После каждого шага выводится содержимое массива. Пример вывода программы heapSort.java:

```
Enter number of items: 10
Random: 81 6 23 38 95 71 72 39 34 53
Heap:   95 81 72 39 53 71 23 38 34 6
..............................................................
                                95
                81                              72
        39              53              71              23
    38      34      6
..............................................................
Sorted: 6 23 34 38 39 53 71 72 81 95
```

Эффективность пирамидальной сортировки

Как упоминалось ранее, пирамидальная сортировка выполняется за время $O(N*\log N)$. Хотя по скорости она слегка уступает быстрой сортировке, ее преимуществом перед быстрой сортировкой является меньшая чувствительность к исходному распределению данных. При некоторых конфигурациях ключей быстрая сортировка замедляется до $O(N^2)$, тогда как пирамидальная сортировка выполняется за время $O(N*\log N)$ независимо от распределения данных.

Итоги

- Приоритетная очередь представляет собой абстрактный тип данных (ADT) с методами для вставки данных и извлечения наибольшего (или наименьшего) элемента.
- Пирамида является эффективной реализацией приоритетной очереди ADT.
- Пирамида обеспечивает извлечение наибольшего элемента и вставку за время $O(N*\log N)$.
- Наибольший элемент всегда находится в корне.
- Пирамида не поддерживает упорядоченный обход данных, поиск элемента с заданным ключом или удаление.
- Пирамида обычно реализуется на базе массива, представляющего полное двоичное дерево. Корневой элемент хранится в ячейке с индексом 0, а последний элемент — в ячейке с индексом $N - 1$.
- Ключ каждого узла меньше ключа его родителя и больше ключей его потомков.
- Вставляемый элемент всегда размещается в первой свободной ячейке массива, а затем смещается вверх до нужной позиции.
- Элемент, извлекаемый из корня, заменяется последним элементом массива, который затем смещается вниз до нужной позиции.
- Смещения вверх и вниз можно представить в виде последовательности перестановок элементов, но они более эффективно реализуются как последовательность операций копирования.
- Пирамида позволяет изменить приоритет произвольного элемента. Сначала изменяется ключ элемента. Если ключ увеличился, то элемент смещается вверх, а если уменьшился, то элемент смещается вниз.
- Пирамида может быть реализована на базе двоичного дерева (не дерева поиска), воспроизводящего структуру пирамиды; такое дерево называется пирамидальным.
- Существуют алгоритмы поиска последнего занятого или первого свободного узла в пирамидальном дереве.
- Пирамидальная сортировка — эффективный алгоритм поиска, выполняемый за время $O(N*\log N)$.
- На концептуальном уровне пирамидальная сортировка состоит из N вставок в пирамиду с последующими N извлечениями.
- Пирамидальную сортировку можно ускорить, применяя алгоритм смещения вниз напрямую к $N/2$ элементам неупорядоченного массива (вместо вставки N элементов).
- Неупорядоченные данные, массив пирамиды и итоговые отсортированные данные могут храниться в одном массиве. Таким образом, пирамидальная сортировка не требует дополнительных затрат памяти.

Вопросы

Следующие вопросы помогут читателю проверить качество усвоения материала. Ответы на них приведены в приложении В.

1. Что означает термин «полный» применительно к двоичным деревьям?
   a) Все необходимые данные вставлены в дерево.
   b) Все уровни заполнены узлами (возможно, кроме нижнего).
   c) Все существующие узлы содержат данные.
   d) Конфигурация узлов соответствует условию пирамиды.
2. Что означает термин «слабая упорядоченность» применительно к двоичным деревьям?
3. Узлы в пирамиде всегда извлекаются из ___________.
4. В ходе смещения вверх в пирамиде, упорядоченной по убыванию:
   a) узел многократно меняется местами со своим родителем, пока не станет больше родителя;
   b) узел многократно меняется местами со своим потомком, пока не станет больше потомка;
   c) узел многократно меняется местами со своим потомком, пока не станет меньше потомка;
   d) узел многократно меняется местами со своим родителем, пока не станет меньше родителя.
5. Пирамида может быть представлена в виде массива, потому что пирамида:
   a) полна;
   b) обладает слабой упорядоченностью;
   c) является двоичным деревом;
   d) удовлетворяет условию пирамиды.
6. Последний узел пирамиды:
   a) всегда является левым потомком;
   b) всегда является правым потомком;
   c) всегда находится на нижнем уровне;
   d) никогда не бывает меньше своего «брата».
7. Пирамида по отношению к приоритетной очереди является тем же, чем ________ является по отношению к стеку.
8. При вставке в пирамиде, упорядоченной по убыванию, используется смещение ______.
9. Пирамидальная сортировка основана на:
   a) извлечении данных из пирамиды и их повторной вставке;
   b) вставке данных в пирамиду и их последующем извлечении;

   c) копировании данных из одной пирамиды в другую;
   d) копировании данных в пирамиду из массива, представляющего пирамиду.

10. Сколько массивов, размер каждого из которых достаточен для хранения всех данных, требуется для сортировки пирамиды?

Упражнения

Упражнения помогут вам глубже усвоить материал этой главы. Программирования они не требуют.

1. Влияет ли порядок вставки данных в пирамиду на конфигурацию узлов? Воспользуйтесь приложением Heap Workshop для проверки.
2. Вставьте в приложении Workshop 10 элементов, упорядоченных по возрастанию, в пустую пирамиду (кнопка `Ins`). Если теперь извлекать эти элементы кнопкой `Rem`, будут ли они извлекаться в противоположном порядке?
3. Вставьте несколько элементов с одинаковыми ключами, затем извлеките их. Можно ли по результатам определить, является ли пирамидальная сортировка устойчивой? Цвет узлов является вторичным элементом данных.

Программные проекты

В этом разделе читателю предлагаются программные проекты, которые укрепляют понимание материала и демонстрируют практическое применение концепций этой главы.

12.1. Преобразуйте программу `heap.java` (см. листинг 12.1), чтобы пирамида была упорядочена по возрастанию, а не по убыванию (То есть корневой узел является наименьшим, а не наибольшим.) Убедитесь в том, что все операции работают правильно.

12.2. В программе `heap.java` метод `insert()` вставляет в пирамиду новый узел и проверяет выполнение условия пирамиды. Напишите метод `toss()`, который помещает новый узел в массив пирамиды без проверки условия. (Например, каждый новый элемент просто помещается в конец массива.) Затем напишите метод `restoreHeap()`, который восстанавливает условие для всей пирамиды. При вставке большого количества элементов многократные вызовы `toss()` с одним итоговым вызовом `restoreHeap()` более эффективны, чем многократные вызовы `insert()`. Чтобы протестировать программу, вставьте в пирамиду несколько элементов, потом добавьте еще несколько при помощи `toss()` и восстановите пирамиду.

12.3. Реализуйте класс `PriorityQ` в программе `priorityQ.java` (см. листинг 4.6) на базе пирамиды (вместо массива). Вы можете использовать класс `Heap` из программы `heap.java` (см. листинг 12.1) без каких-либо изменений. Сделайте очередь упорядоченной по убыванию (с извлечением наибольшего элемента).

12.4. Одной из проблем с реализацией приоритетной очереди в виде пирамиды, реализованной на базе массива, является фиксированный размер массива. Если данные выйдут за границу массива, массив необходимо расширить, как это делалось для хеш-таблиц в программном проекте 11.4 главы 11, «Хеш-таблицы». Проблему можно обойти, реализуя приоритетную очередь на базе обычного дерева двоичного поиска (вместо пирамиды). Такое дерево может разрастаться до неограниченных размеров (если не считать ограничений объема системной памяти).

Начните с класса `Tree` программы `tree.java` (см. листинг 8.1). Внесите изменения в этот класс, соответствующие особенностям приоритетной очереди — добавьте метод извлечения наибольшего элемента `removeMax()`. Для пирамиды метод реализуется просто, для дерева задача усложняется. Как найти наибольший элемент дерева? Нужно ли беспокоиться об обоих потомках извлекаемого узла? Реализация `change()` не является обязательной — в дереве двоичного поиска она легко решается удалением старого элемента и вставкой нового с другим ключом.

Приложение должно работать с классом `PriorityQ`; класс `Tree` должен оставаться невидимым для метода `main()` (допускается только вывод дерева в процессе отладки). Метод вставки и `removeMax()` должны выполняться за время $O(\log N)$.

12.5. Напишите программу, реализующую пирамидальное дерево (реализацию пирамиды на базе дерева), по описанию в тексте главы. Программа должна поддерживать извлечение наибольшего элемента, вставку и изменение ключа.

Глава 13
Графы

Графы принадлежат к числу самых гибких и универсальных структур, используемых в программировании. Как правило, задачи, решаемые при помощи графов, существенно отличаются от задач, которые рассматривались нами ранее. В области типичного хранения данных, скорее всего, графы вам не понадобятся, но в некоторых областях — и притом очень интересных — они оказывают неоценимую помощь.

Наше знакомство с графами будет разделено на две главы. В этой главе рассматриваются алгоритмы невзвешенных графов, приводятся описания некоторых алгоритмов, для представления которых могут использоваться графы, а также двух приложений Workshop для их моделирования. В следующей главе будут рассмотрены более сложные алгоритмы, относящиеся к взвешенным графам.

Знакомство с графами

Граф как структура данных имеет много общего с деревом. Более того, в математическом смысле дерево является частным случаем графа. И все же в программировании графы используются не так, как деревья.

Архитектура структур данных, описанных ранее в книге, определялась применяемыми к ним алгоритмами. Например, двоичное дерево имеет строение, упрощающее поиск данных и вставку новых узлов. Ребра дерева представляют быстрый способ перехода от узла к узлу.

Напротив, строение графа часто определяется физической или абстрактной задачей. Например, узлы графа могут представлять города, а ребра — маршруты авиарейсов между этими городами. Или другой, более абстрактный пример: допустим, имеется некий проект, завершение которого требует выполнения ряда задач. В графе узлы могут представлять задачи, а направленные ребра определяют последовательность их выполнения. В обоих случаях строение графа определяется конкретной ситуацией.

Прежде чем двигаться дальше, необходимо упомянуть, что при описании графов узлы обычно называются *вершинами*. Возможно, это связано с тем, что терминология графов сформировалась намного раньше — несколько столетий назад в области математики. Деревья в большей степени привязаны к практическому программированию. Тем не менее оба термина являются более или менее равноправными.

Определения

На рис. 13.1, *а* изображена упрощенная карта автострад в окрестностях Сан-Хосе (штат Калифорния). На рис. 13.1, *б* представлен граф, моделирующий дорожную сеть.

На графе кружки соответствуют дорожным развязкам, а прямые линии, которыми эти кружки соединяются, — дорожным сегментам. Кружки являются *вершинами*, а линии — *ребрами* графа. Вершины обычно каким-то образом помечаются — часто для пометки используются буквы, как в этом случае. Каждое ребро ограничивается двумя вершинами, находящимися на его концах.

Граф не пытается представить географические координаты отдельных городов; он только моделирует отношения между вершинами и ребрами, то есть какие ребра соединяют те или иные вершины. Граф не имеет отношения к физическим расстояниям или направлениям. Кроме того, одно ребро может представлять несколько автострад — как, например, ребро от вершины I к H, представляющее дороги 101, 84 и 280. Важен сам факт соединения одной развязки с другой (или его отсутствия), а не конкретные дороги.

Смежность

Две вершины называются *смежными*, если они соединены одним ребром. Так, на рис. 13.1 вершины I и G являются смежными, а вершины I и F — нет. Вершины, смежные с некоторой вершиной, называются ее *соседями*. Например, соседями G являются I, H и F.

Пути

Путь представляет собой последовательность вершин. На рис. 13.1 показан путь от вершины B к вершине J, проходящий через вершины A и E; этот путь можно обозначить BAEJ. Между двумя вершинами может существовать более одного пути; например, от B к J также ведет путь BCDJ.

Связные графы

Граф называется *связным*, если от каждой вершины к любой другой вершине ведет хотя бы один путь (рис. 13.2, *а*). Но если «отсюда дотуда не добраться» (как говорят фермеры в Вермонте городским пижонам, спрашивающим дорогу), граф становится *несвязным* — пример показан на рис. 13.2, *б*.

Несвязный граф состоит из нескольких областей. На рис. 13.2, *б* одна из таких областей состоит из вершин A и B, а другая — из вершин C и D.

Для простоты алгоритмы, описанные в этой главе, рассчитаны на работу со связными графам или отдельными областями несвязных графов. Как правило, после незначительных изменений эти алгоритмы смогут работать и с несвязными графами.

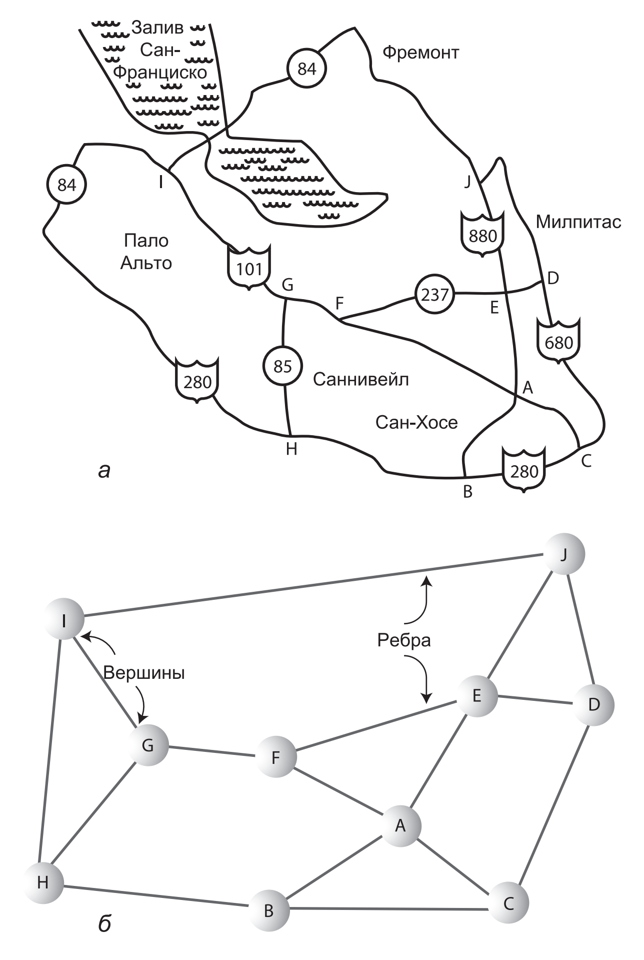


Рис. 13.1. Географическая карта (*а*) и графы (*б*)

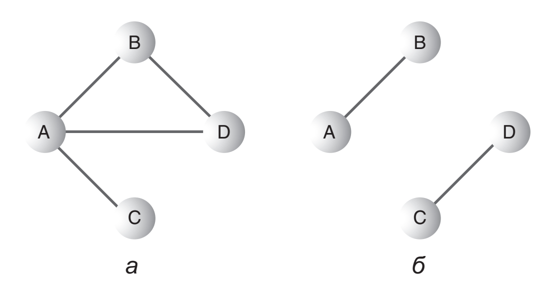


Рис. 13.2. Связные (*а*) и несвязные (*б*) графы

Направленные и взвешенные графы

На рис. 13.1 и 13.2 изображены *ненаправленные* графы. Иначе говоря, ребра таких графов не имеют направления; перемещения по ним возможны в обоих направлениях, то есть алгоритм может перейти как от вершины A к вершине B, так и от

вершины B к вершине A. Ненаправленные графы хорошо моделируют дорожную сеть, потому что по дорогам обычно можно ездить в обоих направлениях.

Однако графы также часто используются для моделирования ситуаций, в которых перемещение по ребру возможно только в одном направлении — от A к B, но не от B к A, как на улице с односторонним движением. Такие графы называются *направленными* (или *ориентированными*). Разрешенное направление обычно обозначается стрелкой на конце ребра.

На некоторых графах ребрам присваиваются *веса* — числа, представляющие физическое расстояние между двумя вершинами, время перехода от вершины к вершине или затраты на такой переход (скажем, в примере с авиарейсами). Такие графы, называемые *взвешенными*, подробно рассматриваются в следующей главе. А эта глава начинается с описания простых ненаправленных, невзвешенных графов; потом мы перейдем к направленным невзвешенным графам.

Приведенные определения ни в коей мере не исчерпывают всей терминологии, относящейся к графам. Другие термины будут вводиться по мере изложения материала.

Немного истории

Одним из первых математиков, занимавшихся теорией графов, был Леонард Эйлер (начало XVIII века). В частности, он решил знаменитую задачу с «кенигсбергскими мостами» — в городе Кенигсберг был остров с семью мостами (рис. 13.3, *а*). В популярной задаче требовалось найти путь пересечения всех семи мостов, в котором каждый мост проходился бы только один раз. Мы не будем описывать решение Эйлера; он доказал, что такого пути не существует. Для нас важно то, что его решение было основано на представлении задачи в виде графа. Вершины графа моделировали участки земли, а ребра — мосты (рис. 13.3, *б*). Возможно, это был первый случай использования графа для моделирования задачи из реального мира.

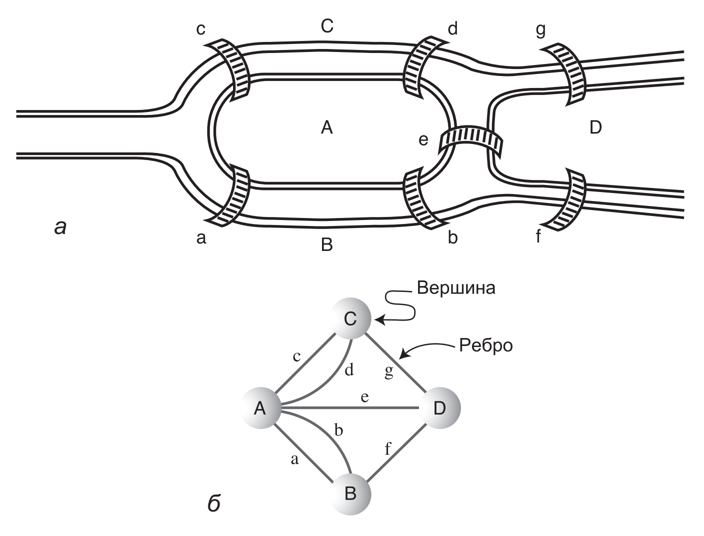


Рис. 13.3. Кенигсбергские мосты: *а* — карта; *б* — граф

Представление графа в программе

До изобретения компьютеров Эйлер и другие математики изучали графы с абстрактной точки зрения. Нас больше интересует практическое представление графов в компьютерных программах. Какая структура данных лучше подойдет для моделирования графа? Начнем с вершин, а затем перейдем к ребрам.

Вершины

В очень абстрактной программе для работы с графом можно пронумеровать вершины от 0 до $N - 1$ (где N — количество вершин). Для хранения вершин переменные вообще не нужны, потому что их полезность зависит от связей с другими вершинами.

Однако на практике вершины обычно представляют реально существующие объекты, а объекты описываются полями данных. Например, если вершина представляет город в модели сети воздушного сообщения, в ней может храниться название города, высота над уровнем моря, координаты и т. д. Таким образом, для представления вершин обычно удобно использовать объекты соответствующего класса. В наших примерах программ для каждой вершины хранится только буквенная метка (например, A) и флаг, используемый поисковыми алгоритмами (об этом ниже). Определение класса вершины графа `Vertex` выглядит так:

```
class Vertex
   {
   public char label;        // Метка (например, 'A')
   public boolean wasVisited;

   public Vertex(char lab)   // Конструктор
      {
      label = lab;
      wasVisited = false;
      }
   }  // Конец класса Vertex
```

Объекты вершин можно хранить в массиве и обращаться к ним по индексу. В наших примерах они будут храниться в массиве с именем `vertexList`. Для хранения вершин также можно использовать список или другую структуру данных. Впрочем, структура для хранения вершин выбирается только для удобства. Она никак не связана со способом соединения вершин ребрами. Для моделирования ребер графа понадобится другой механизм.

Ребра

Как было показано в главе 9, «Красно-черные деревья», в программах могут использоваться разные способы представления деревьев. В основном мы изучали представление, в котором каждый узел содержит ссылки на своих потомков, но также было рассмотрено представление на базе массива, в котором позиция узла в массиве определяет его связи с другими узлами. В главе 12, «Пирамиды», была представлена возможность использования массива для представления особой разновидности деревьев — так называемых пирамид.

Однако граф обычно не имеет такой жесткой структуры, как дерево. В двоичном дереве каждый узел имеет не более двух потомков, а в графе каждая вершина может быть соединена с произвольным количеством других вершин. Например, на рис. 13.2, *а* вершина A соединена с тремя другими вершинами, а вершина C соединена только с одной.

Для моделирования подобной нежесткой структуры нужен другой способ представления ребер. При работе с графами обычно применяются две структуры: *матрица смежности* и *список смежности*. (Напомним, что вершины называются смежными, если они соединены одним ребром.)

Матрица смежности

Матрица смежности представляет собой двумерный массив, элементы которого обозначают наличие связи между двумя вершинами. Если граф содержит N вершин, то матрица смежности представляет собой массив $N \times N$. В таблице 13.1 приведена матрица смежности для графа на рис. 13.2, *а*.

Таблица 13.1. Матрица смежности

	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

Заголовками как строк, так и столбцов являются метки вершин. Ребро между двумя вершинами обозначается 1; при отсутствии ребра элемент матрицы равен 0. (Также можно использовать логические значения `true/false`.) Как видно из таблицы, вершина A является смежной со всеми тремя остальными вершинами, вершина B — с вершинами A и D, вершина C — только с A, а вершина D — с A и B. В этой матрице «соединение» вершины с самой собой обозначается 0, поэтому диагональ, идущая из левого верхнего в правый нижний угол (от A–A до D–D), заполнена нулями. Элементы диагонали не несут полезной информации, поэтому в них с таким же успехом можно записать 1, если это окажется удобнее в вашей программе.

Обратите внимание: треугольная часть матрицы над диагональю является зеркальным отражением части, находящейся под ней; оба треугольника содержат одни и те же данные. Подобная избыточность может показаться неэффективной, но в большинстве языков программирования не существует удобных средств для представления треугольных массивов, поэтому проще смириться с избыточностью. Соответственно при добавлении ребра в граф в матрице смежности необходимо изменить два элемента вместо одного.

Список смежности

В другом способе представления ребер графа используется список смежности (речь идет о связанных списках, рассматривавшихся в главе 5). Вообще говоря,

список смежности скорее является массивом списков (а иногда списком списков). Каждый отдельный список содержит информацию о том, какие вершины являются смежными по отношению к заданной. В таблице 13.2 приведены списки смежности для графа на рис. 13.2, *а*.

Таблица 13.2. Списки смежности

Вершина	**Список смежных вершин**
A	B—>C—>D
B	A—>D
C	A
D	A—>B

В этой таблице знаком —> обозначается связь в списке. Элементами списка являются вершины. В нашем примере вершины каждого списка упорядочены в алфавитном порядке, хотя на самом деле это не обязательно. Не путайте содержимое списков смежности с путями. Список смежности показывает, какие вершины являются смежными по отношению к заданной (то есть находятся от нее на расстоянии одного ребра); он не является описанием пути между вершинами.

Позднее мы разберемся, в каких ситуация предпочтительнее использовать матрицу смежности, а когда лучше выбрать список смежности. Во всех приложениях Workshop этой главы используется матрица смежности, хотя в некоторых случаях списковое решение оказывается более эффективным.

Добавление вершин и ребер в граф

Чтобы включить в граф новую вершину, создайте новый объект вершины оператором `new` и вставьте его в массив вершин `vertexList`. В реальной программе объект вершины может содержать много полей данных, но в наших примерах для простоты используется только один символ. Таким образом, создание вершины выглядит примерно так:

```
vertexList[nVerts++] = new Vertex('F');
```

Команда вставляет в граф вершину F. В переменной `nVerts` хранится текущее количество вершин в графе. Способ добавления ребра зависит от выбранного представления (матрица смежности или списки смежности). Допустим, в программе используется матрица смежности, и в граф добавляется ребро между вершинами 1 и 3. Эти числа соответствуют индексам массива `vertexList`, в котором хранятся вершины. При создании матрица смежности `adjMat` заполняется нулями. Чтобы вставить в нее ребро, выполните следующие команды:

```
adjMat[1][3] = 1;
adjMat[3][1] = 1;
```

В программе со списком смежности в список 3 добавляется вершина 1, а в список вершины 1 добавляется вершина 3.

Класс Graph

Класс Graph содержит методы для создания списка вершин и матрицы смежности, а также для добавления вершин и ребер в объект Graph:

```
class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // Массив вершин
   private int adjMat[][];      // Матрица смежности
   private int nVerts;          // Текущее количество вершин
// -------------------------------------------------------------
   public Graph()               // Конструктор
      {
      vertexList = new Vertex[MAX_VERTS];
                                // Матрица смежности
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // Матрица смежности
         for(int k=0; k<MAX_VERTS; k++)   // заполняется нулями
            adjMat[j][k] = 0;
      }
// -------------------------------------------------------------
   public void addVertex(char lab)    // В аргументе передается метка
      {
      vertexList[nVerts++] = new Vertex(lab);
      }
// -------------------------------------------------------------
   public void addEdge(int start, int end)
      {
      adjMat[start][end] = 1;
      adjMat[end][start] = 1;
      }
// -------------------------------------------------------------
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);
      }
// -------------------------------------------------------------
   }  // Конец класса Graph
```

В классе Graph вершины определяются своим индексом в списке vertexList. Большинство приведенных методов уже рассматривалось ранее. Вывод вершины сводится к выводу ее символьной метки.

Информация, хранящаяся в матрице смежности (или списке смежности), является локальной для конкретной вершины, а именно она сообщает, какие вершины соединены одним ребром с заданной вершиной. Чтобы получить ответы на более глобальные вопросы о взаимном расположении вершин, необходимо использовать специальные алгоритмы. Начнем с обхода.

Обход

Одной из основных операций, выполняемых с графами, является определение всех вершин, достижимых от заданной вершины. Представьте, что вы пытаетесь определить, до каких городов в США можно добраться пассажирским поездом от Канзас-Сити (с возможными пересадками). До одних городов добраться можно. Другие города недоступны, потому что в них нет железнодорожного сообщения. Третьи недоступны даже при наличии железной дороги, например из-за того, что их узкоколейное полотно не стыкуется со стандартной шириной колеи вашей железнодорожной сети.

Или другая ситуация, в которой тоже может потребоваться найти все вершины, к которым можно перейти от заданной вершины. Представьте, что вы проектируете печатную плату вроде тех, которые установлены в вашем компьютере. На плате размещаются различные компоненты — в основном микросхемы, контакты которых вставляются в отверстия на плате. Микросхемы закрепляются посредством пайки, а электрическое соединение их контактов с другими контактами осуществляется при помощи *дорожек* — тонких металлических полосок, проходящих по поверхности платы (рис. 13.4).

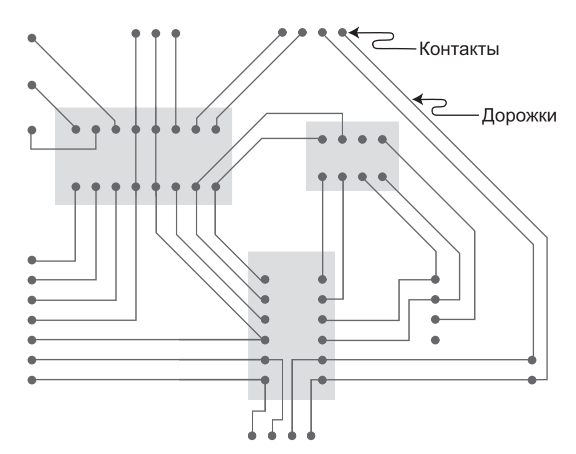


Рис. 13.4. Контакты и дорожки на печатной плате

На графе контакты могут представляться вершинами, а дорожки — ребрами. На печатной плате обычно имеется много электрических цепей, не связанных друг с другом, поэтому граф ни в коем случае не обязан быть связным. А следовательно, в процессе проектирования будет полезно построить граф и воспользоваться им для определения того, какие контакты подключены к текущей цепи.

Предположим, такой граф построен. Теперь нужен алгоритм, который начинается от заданной вершины и систематически перемещается по ребрам к другим вершинам так, что после завершения его работы будут посещены все вершины, соединенные с исходной. Как в главе 8, «Двоичные деревья», под «посещением» понимается выполнение с вершиной некоторой операции, например вывод.

Существует два основных способа обхода графов: *обход в глубину* и *обход в ширину*. Оба способа в конечном итоге обеспечивают перебор всех соединенных вершин. Обход в глубину реализуется на базе стека, а обход в ширину реализуется на базе очереди. Как вы вскоре убедитесь, это приводит к обходу вершин графа в разном порядке.

Обход в глубину

Алгоритм обхода в глубину хранит в стеке информацию о том, куда следует вернуться при достижении «тупика». Мы рассмотрим пример, немного поэкспериментируем с приложением GraphN Workshop, а потом разберем код конкретной реализации.

Пример

Для объяснения принципов обхода в глубину будет использован граф на рис. 13.5. Цифры на графе обозначают порядок посещения вершин.

Обход в глубину начинается с выбора отправной точки — в нашем примере это вершина A. Затем алгоритм выполняет три операции: посещает вершину, заносит ее в стек и помечает для предотвращения повторных посещений.

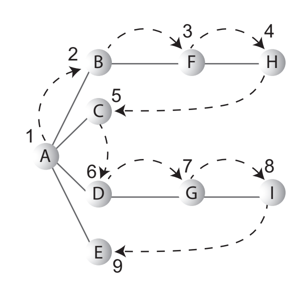


Рис. 13.5. Обход в глубину

Затем алгоритм переходит к любой вершине, смежной с A, которая еще не посещалась ранее. Будем считать, что вершины выбираются в алфавитном порядке; значит, это будет вершина B. Алгоритм посещает B, помечает вершину и заносит в стек.

Что теперь? Текущей вершиной является B, поэтому алгоритм делает то же, что и прежде: он переходит к смежной вершине, которая еще не посещалась ранее. В нашем примере это вершина F. Назовем этот процесс правилом 1.

ПРАВИЛО 1

Посетить смежную вершину, не посещавшуюся ранее, пометить ее и занести в стек.

Применение правила 1 снова приводит к вершине H. Однако на этот раз необходимо сделать что-то другое, потому что в графе не осталось непосещенных вершин, смежных с H. На помощь приходит правило 2.

ПРАВИЛО 2

Если выполнение правила 1 невозможно, извлечь вершину из стека.

В соответствии с этим правилом вершина H извлекается из стека, в результате чего алгоритм возвращается к F. У вершины F тоже нет непосещенных смеж-

ных вершин; она извлекается из стека. То же происходит с вершиной B. Теперь в стеке остается только вершина A. Однако у A имеются непосещенные смежные вершины, поэтому алгоритм посещает следующую вершину C. Но поскольку эта вершина тоже является тупиковой, она извлекается из стека, и алгоритм возвращается к A. Далее алгоритм посещает D, G и I, а затем извлекает их при достижении тупика в I. Происходит возврат к A. Алгоритм посещает вершину E и снова возвращается к A.

Но на этот раз у вершины A не осталось непосещенных соседей, и она извлекается из стека. В стеке не остается элементов, и вступает в действие правило 3.

ПРАВИЛО 3

Если выполнение правил 1 и 2 невозможно, обход закончен.

В таблице 13.3 представлено содержимое стека на разных стадиях этого процесса для рис. 13.5.

Таблица 13.3. Содержимое стека при обходе в глубину

Событие	Стек
Посещение A	A
Посещение B	AB
Посещение F	ABF
Посещение H	ABFH
Извлечение H	ABF
Извлечение F	AB
Извлечение B	A
Посещение C	AC
Извлечение C	A
Посещение D	AD
Посещение G	ADG
Посещение I	ADGI
Извлечение I	ADG
Извлечение G	AD
Извлечение D	A
Посещение E	AE
Извлечение E	A
Извлечение A	
Обход завершен	

Содержимое стека описывает путь от исходной вершины до текущей. Удаляясь от исходной вершины, алгоритм заносит вершины в стек, а при возвращении

к ней — извлекает из стека. В нашем примере вершины будут посещаться в порядке ABFHCDGIE.

Таким образом, алгоритм обхода в глубину стремится как можно быстрее удалиться от исходной вершины и возвращается к ней только при достижении тупика. Если понимать под «глубиной» расстояние от исходной вершины, становится понятно, откуда взялось название алгоритма.

Аналогия

Обход в глубину можно сравнить с поиском пути в лабиринте. Лабиринт состоит из узких проходов (ребер) и перекрестков (вершин), на которых они встречаются друг с другом.

Допустим, некто заблудился в лабиринте. Он знает, что выход существует, и пытается систематично обойти весь лабиринт в поисках выхода. К счастью, у странника имеется клубок веревки и фломастер. Он начинает с ближайшего перекрестка и идет по случайно выбранному проходу, разматывая веревку. На следующем перекрестке он снова сворачивает на другой случайно выбранный проход и т. д., пока не окажется в тупике. Странник возвращается, сматывая веревку, пока не окажется у последнего перекрестка. Здесь он помечает тупиковый путь, чтобы не заходить в него в будущем, и опробует другой проход. Когда все пути от текущего перекрестка будут проверены, странник возвращается к предыдущему перекрестку, и все повторяется снова. Веревка в этом примере является аналогом стека: она используется для «запоминания» пути к текущей позиции.

Приложение GraphN Workshop и обход в глубину

Кнопка `DFS` в приложении GraphN Workshop позволяет поэкспериментировать с обходом в глубину.

Запустите приложение. В исходном состоянии граф не содержит ни вершин, ни ребер — рабочая область представляет собой пустой прямоугольник. Вершины создаются двойным щелчком в нужной позиции. Первой вершине автоматически присваивается метка A, второй — метка B и т. д. Цвета вершин выбираются случайным образом. Чтобы создать ребро, перетащите указатель мыши между вершинами. На рис. 13.6 изображен граф с рис. 13.5, созданный в приложении.

Возможность удаления отдельных ребер или вершин не предусмотрена, так что если вы допустите ошибку, придется начать все заново — кнопка `New` уничтожает все созданные вершины и ребра (после подтверждения пользователя). Кнопка `View` выводит матрицу смежности созданного графа (рис. 13.7). Повторное нажатие кнопки `View` возвращает режим отображения графа.

Алгоритм обхода в глубину запускается кнопкой `DFS`. Вам будет предложено щелкнуть (один раз!) на вершине, с которой должен начаться обход.

Возьмите за образец граф на рис. 13.6 или создайте более или менее сложный граф по своему усмотрению. После непродолжительных экспериментов вы сможете предсказать, что алгоритм сделает на следующем шаге (если граф не отличается особой сложностью).

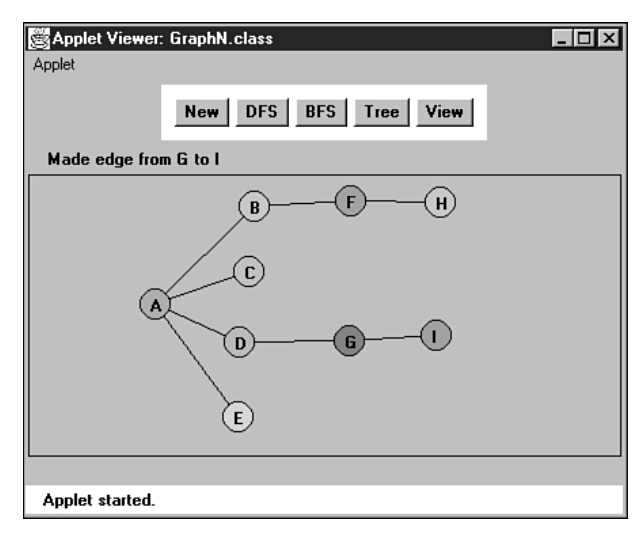


Рис. 13.6. Приложение GraphN Workshop

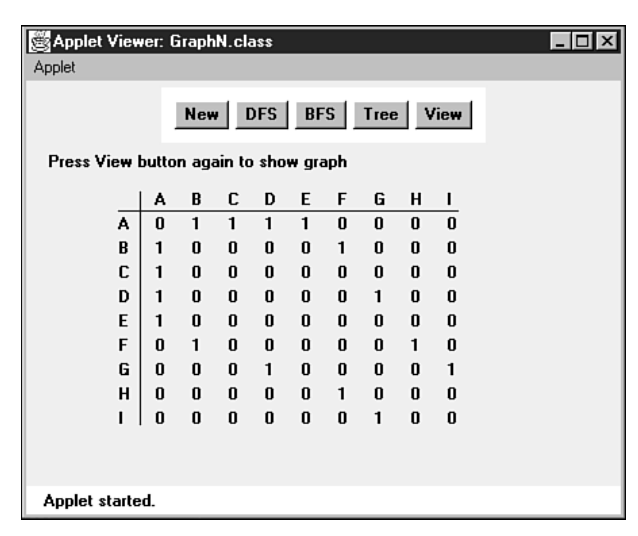


Рис. 13.7. Вывод матрицы смежности в GraphN Workshop

При выполнении для несвязного графа алгоритм обходит только вершины, соединенные с исходной вершиной.

Реализация на языке Java

Центральное место в алгоритме обхода в глубину занимает поиск вершин, смежных по отношению к заданной и ранее не посещавшихся. Как решить эту задачу? При помощи матрицы смежности. Программа просматривает строку заданной вершины и отбирает столбцы, содержащие 1; номер столбца определяет номер смежной вершины. Далее программа проверяет, посещалась ли вершина ранее. Если вершина

не посещалась, значит, мы нашли искомое — следующую вершину для посещения. Если в строке не осталось ни одной вершины с 1 в ячейке матрицы (смежной), для которой не был бы установлен флаг посещения, значит, непосещенных вершин, смежных по отношению к данной, не осталось. Эта логика запрограммирована в методе getAdjUnvisitedVertex():

```
// Метод возвращает непосещенную вершину, смежную по отношению к v
public int getAdjUnvisitedVertex(int v)
   {
   for(int j=0; j<nVerts; j++)
      if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
         return j;              // Возвращает первую найденную вершину
   return -1;                   // Таких вершин нет
   }
```

Теперь у нас есть все необходимое для написания метода dfs() класса Graph, выполняющего обход в глубину. Код наглядно реализует три правила, перечисленных ранее. Цикл продолжает выполняться, пока в стеке остается хотя бы один элемент. На каждой итерации:

1) проверяется элемент на вершине стека методом peek();
2) делается попытка найти непосещенного соседа этой вершины;
3) если поиск окажется неудачным, элемент извлекается из стека;
4) если вершина будет найдена, алгоритм посещает ее и заносит в стек.

Код метода dfs():

```
public void dfs()  // Обход в глубину
   {                                 // Алгоритм начинает с вершины 0
   vertexList[0].wasVisited = true;  // Пометка
   displayVertex(0);                 // Вывод
   theStack.push(0);                 // Занесение в стек

   while( !theStack.isEmpty() )      // Пока стек не опустеет
      {
      // Получение непосещенной вершины, смежной к текущей
      int v = getAdjUnvisitedVertex( theStack.peek() );
      if(v == -1)                    // Если такой вершины нет,
         theStack.pop();             // элемент извлекается из стека
      else                           // Если вершина найдена
         {
         vertexList[v].wasVisited = true;  // Пометка
         displayVertex(v);                 // Вывод
         theStack.push(v);                 // Занесение в стек
         }
      }

      // Стек пуст, работа закончена
      for(int j=0; j<nVerts; j++)    // Сброс флагов
         vertexList[j].wasVisited = false;
   }
```

В конце метода dfs() сбрасываются флаги wasVisited, чтобы dfs() можно было снова вызвать в будущем. Стек к этому моменту уже пуст и очищать его не нужно.

Теперь у нас имеются все необходимые компоненты класса Graph. Следующий фрагмент создает объект графа, включает в него несколько вершин и ребер, а затем выполняет обход в глубину:

```
Graph theGraph = new Graph();
theGraph.addVertex('A');    // 0  (исходная вершина)
theGraph.addVertex('B');    // 1
theGraph.addVertex('C');    // 2
theGraph.addVertex('D');    // 3
theGraph.addVertex('E');    // 4

theGraph.addEdge(0, 1);     // AB
theGraph.addEdge(1, 2);     // BC
theGraph.addEdge(0, 3);     // AD
theGraph.addEdge(3, 4);     // DE

System.out.print("Visits: ");
theGraph.dfs();             // Обход в глубину
System.out.println();
```

Граф, построенный этим фрагментом, изображен на рис. 13.8. Результат выполнения выглядит так:

```
Visits: ABCDE
```

Попробуйте изменить этот код, чтобы он строил другой граф по вашему выбору; запустите его и проверьте, правильно ли выполняется обход в глубину.

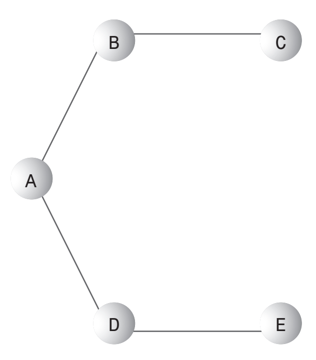


Рис. 13.8. Граф, используемый в программах dfs.java и bfs.java

Программа dfs.java

В листинге 13.1 приведен код программы dfs.java с методом dfs(). В ней используется измененная версия класса StackX из главы 4, «Стеки и очереди».

Листинг 13.1. Программа dfs.java

```
// dfs.java
// Обход в глубину
// Запуск программы: C>java DFSApp
////////////////////////////////////////////////////////////////
class StackX
   {
   private final int SIZE = 20;
   private int[] st;
   private int top;
// ------------------------------------------------------------
   public StackX()           // Конструктор
      {
      st = new int[SIZE];    // Создание массива
      top = -1;
      }
```

```
// -------------------------------------------------------------
   public void push(int j)   // Размещение элемента в стеке
      { st[++top] = j; }
// -------------------------------------------------------------
   public int pop()          // Извлечение элемента из стека
      { return st[top--]; }
// -------------------------------------------------------------
   public int peek()         // Чтение с вершины стека
      { return st[top]; }
// -------------------------------------------------------------
   public boolean isEmpty()  // true, если стек пуст
      { return (top == -1); }
// -------------------------------------------------------------
}  // Конец класса StackX
////////////////////////////////////////////////////////////////
class Vertex
   {
   public char label;        // метка (например, 'A')
   public boolean wasVisited;
// -------------------------------------------------------------
   public Vertex(char lab)   // Конструктор
      {
      label = lab;
      wasVisited = false;
      }
// -------------------------------------------------------------
   }  // Конец класса Vertex
////////////////////////////////////////////////////////////////
class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // Список вершин
   private int adjMat[][];      // Матрица смежности
   private int nVerts;          // Текущее количество вершин
   private StackX theStack;
// -------------------------------------------------------------
   public Graph()               // Конструктор
      {
      vertexList = new Vertex[MAX_VERTS];
                                // Матрица смежности
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // Матрица смежности
         for(int k=0; k<MAX_VERTS; k++)   // заполняется нулями
            adjMat[j][k] = 0;
      theStack = new StackX();
      }
// -------------------------------------------------------------
```

продолжение ⇨

Листинг 13.1 (*продолжение*)

```
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
      }
// -------------------------------------------------------------
   public void addEdge(int start, int end)
      {
      adjMat[start][end] = 1;
      adjMat[end][start] = 1;
      }
// -------------------------------------------------------------
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);
      }
// -------------------------------------------------------------
   public void dfs()  // Обход в глубину
      {                                 // Алгоритм начинает с вершины 0
      vertexList[0].wasVisited = true;  // Пометка
      displayVertex(0);                 // Вывод
      theStack.push(0);                 // Занесение в стек

      while( !theStack.isEmpty() )      // Пока стек не опустеет
         {
         // Получение непосещенной вершины, смежной к текущей
         int v = getAdjUnvisitedVertex( theStack.peek() );
         if(v == -1)                    // Если такой вершины нет,
            theStack.pop();             // элемент извлекается из стека
         else                           // Если вершина найдена
            {
            vertexList[v].wasVisited = true;  // Пометка
            displayVertex(v);                 // Вывод
            theStack.push(v);                 // Занесение в стек
            }
         }

      // Стек пуст, работа закончена
      for(int j=0; j<nVerts; j++)    // Сброс флагов
         vertexList[j].wasVisited = false;
      }
// -------------------------------------------------------------
   // Метод возвращает непосещенную вершину, смежную по отношению к v
   public int getAdjUnvisitedVertex(int v)
      {
      for(int j=0; j<nVerts; j++)
         if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
            return j;               // Возвращает первую найденную вершину
      return -1;                    // Таких вершин нет
      }
```

```
// -------------------------------------------------------------
   }  // Конец класса Graph
////////////////////////////////////////////////////////////////
class DFSApp
   {
   public static void main(String[] args)
      {
      Graph theGraph = new Graph();
      theGraph.addVertex('A');    // 0  (исходная вершина)
      theGraph.addVertex('B');    // 1
      theGraph.addVertex('C');    // 2
      theGraph.addVertex('D');    // 3
      theGraph.addVertex('E');    // 4

      theGraph.addEdge(0, 1);     // AB
      theGraph.addEdge(1, 2);     // BC
      theGraph.addEdge(0, 3);     // AD
      theGraph.addEdge(3, 4);     // DE

      System.out.print("Visits: ");
      theGraph.dfs();             // Обход в глубину
      System.out.println();
      }
   }  // Конец класса DFSApp
////////////////////////////////////////////////////////////////
```

Обход в глубину в программировании игр

Обход в глубину часто используется в программировании игр (и сходных ситуаций в реальном мире). В типичной игре существует выбор между несколькими действиями. Каждый вариант приводит к дополнительному набору вариантов, те ведут к своим вариантам и т. д. — постоянно расширяющийся граф вариантов. Точка выбора соответствует вершине, а выбранный вариант — ребру графа, которое ведет к следующей точке выбора.

Вспомните игру «крестики-нолики». Первый игрок может сделать один из 9 возможных ходов. Его противник отвечает одним из 8 возможных ходов и т. д. Каждый ход ведет к следующей группе вариантов противника, каждый из которых ведет к группе вариантов для вас и т. д., пока не будет заполнена последняя клетка.

Как выбрать ход в текущей позиции? Можно мысленно представить себе ход, рассмотреть возможные ответы противника, потом ваши ответы на них и т. д. Ход выбирается в зависимости от того, какой из путей ведет к оптимальному результату. В простых играх вроде «крестики-нолики» количество возможных ходов ограничено, что позволяет проанализировать все пути до конца игры. Полный анализ путей определяет начальный ход. Ситуация может быть представлена в виде графа с одним узлом, представляющим первый ход. Узел соединен с восемью узлами, представляющими возможные ответы противника; каждый из этих узлов соединяется с семью узлами, представляющими ваши ответы и т. д. Каждый путь

от начального узла к конечному состоит из девяти узлов. Чтобы выполнить полный анализ, необходимо построить девять графов, по одному для каждого начального хода.

Даже в этой простой игре количество путей оказывается неожиданно большим. Если забыть об упрощениях, связанных с симметрией, девять графов образуют 9*8*7*6*5*4*3*2*1 путей. Результат равен 9! (факториал 9), или 362 880. В такой игре, как шахматы, где количество возможных ходов намного больше, даже самые мощные компьютеры (такие, как «Deep Blue» фирмы IBM) не способны «видеть» позицию до конца партии. Они только проходят путь до определенной глубины, а затем оценивают позицию на доске и определяют, выглядит ли она более перспективной по сравнению с другими вариантами.

Естественный способ анализа таких ситуаций в программе основан на использовании обхода в глубину. В каждом узле алгоритм принимает решение, что делать дальше, как в методе `getAdjUnvisitedVertex()` программы `dfs.java` (см. листинг 13.1). Если в графе еще остались непосещенные узлы (точки выбора), алгоритм заносит текущий узел в стек и переходит к следующему. Если в некоторой позиции следующий ход невозможен (`getAdjUnvisitedVertex()` возвращает –1), алгоритм «отступает назад», извлекая узел из стека, и проверяет, не осталось ли в этой позиции неисследованных вариантов.

Последовательность ходов хорошо представляется в виде дерева, узлы которого соответствуют отдельным ходам. Первый ход представлен корневым узлом. В «крестиках-ноликах» после первого хода возможно восемь вторых ходов, каждый из которых представлен узлом, соединенным с корнем. После восьми вторых ходов возможно семь третьих ходов, представленных узлами, соединенных с узлами второго кода. Построенное таким образом дерево состоит из 9! возможных путей от корня к листовым узлам. Оно называется *деревом игры*.

На самом деле количество ветвей в дереве игры несколько меньше максимального, потому что выигрыш часто достигается до заполнения всех клеток. Но несмотря на это, полный анализ игры «крестики-нолики» остается относительно сложным, а ведь эта игра очень проста по сравнению с многими другими (скажем, шахматами).

Лишь часть путей дерева игры ведет к выигрышу. Другие пути ведут к победе противника. При достижении такого результата алгоритм должен отступить к предыдущему узлу и проверить другой путь. Анализ дерева продолжается до тех пор, пока не будет найден путь к успешному результату. Далее остается сделать первый ход на этом пути.

Обход в ширину

Как было показано ранее, алгоритм обхода в глубину старается как можно быстрее удалиться от исходной вершины на максимальное расстояние. При обходе в ширину, напротив, алгоритм стремится держаться как можно ближе к исходной вершине. Он посещает все вершины, смежные с исходной, и только после этого отходит дальше. Такая разновидность обхода реализуется на базе очереди (вместо стека).

Пример

На рис. 13.9 изображен тот же граф, что и на рис. 13.5, но на этот раз применяется обход в ширину. Как и прежде, цифрами обозначается порядок посещения вершин.

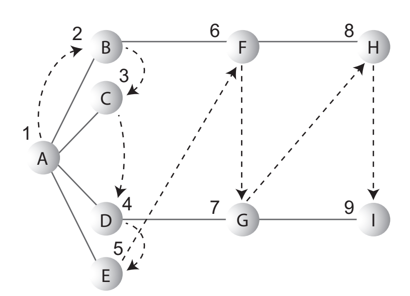


Рис. 13.9. Обход в ширину

Обход начинается с вершины A; алгоритм посещает ее и делает текущей. Далее применяются следующие правила.

ПРАВИЛО 1

Посетить следующую вершину, не посещавшуюся ранее, смежную с текущей вершиной, пометить ее и занести в очередь.

ПРАВИЛО 2

Если выполнение правила 1 невозможно, извлечь вершину из очереди и сделать ее текущей вершиной.

ПРАВИЛО 3

Если выполнение правил 1 и 2 невозможно, обход закончен.

Таким образом, сначала посещаются все вершины, смежные с A. Алгоритм вставляет каждую вершину в очередь при посещении. Так посещаются вершины A, B, C, D и E. На этой стадии очередь (от начала к концу) содержит вершины BCDE.

Других непосещенных и смежных с A вершин не осталось, поэтому алгоритм извлекает B из очереди и ищет вершины, смежные с B. Алгоритм находит вершину F и вставляет ее в очередь. Других непосещенных вершин, смежных с B, не осталось, поэтому вершина C извлекается из очереди. И снова непосещенных смежных вершин нет, алгоритм извлекает D из очереди и посещает G. У D не осталось других смежных непосещенных вершин, из очереди извлекается E. Теперь в очереди остались вершины FG. Алгоритм извлекает F и посещает H, затем извлекает G и посещает I.

В результате очередь содержит вершины HI. Но когда алгоритм извлекает каждую из них и не находит смежных непосещенных вершин, очередь остается пустой, и работа алгоритма завершается. В таблице 13.4 приведено содержимое очереди на разных стадиях процесса обхода.

Таблица 13.4. Содержимое очереди при обходе в ширину

Событие	Очередь
Посещение A	
Посещение B	B
Посещение C	BC
Посещение D	BCD
Посещение E	BCDE
Извлечение B	CDE
Посещение F	CDEF
Извлечение C	DEF
Извлечение D	EF
Посещение G	EFG
Извлечение E	FG
Извлечение F	G
Посещение H	GH
Извлечение G	H
Посещение I	HI
Извлечение H	I
Извлечение I	
Обход завершен	

В каждый промежуточный момент в очереди хранятся вершины, которые были посещены в процессе обхода, но соседи которых не были полностью исследованы. (Сравните с обходом в ширину, при котором в стеке хранится маршрут от исходной вершины к текущей.) Узлы посещаются в порядке ABCDEFGHI.

Приложение GraphN Workshop и обход в ширину

Поэкспериментируйте с обходом в ширину в приложении GraphN Workshop (кнопка `BFS`). Как и в предыдущем случае, вы можете взять за образец рис. 13.9 или построить собственный граф.

Обратите внимание на сходство и различие между обходом в ширину и обходом в глубину.

Последовательность обхода в ширину можно сравнить с кругами, расходящимися по воде от брошенного камня, или для читателей, сведущих в эпидемиологии, с распространением вируса гриппа от города к городу. Сначала посещаются все вершины, соединенные с исходной, затем все вершины, удаленные от исходной на два ребра и т. д.

Реализация на языке Java

Метод bfs() класса Graph похож на метод dfs(), но стек в нем заменен очередью, а один цикл — вложенными циклами. Внешний цикл проверяет наличие элементов в очереди, а внутренний последовательно перебирает каждого непосещенного соседа текущей вершины. Код метода:

```
public void bfs()                    // Обход в ширину
   {                                 // Алгоритм начинает с вершины 0
   vertexList[0].wasVisited = true; // Пометка
   displayVertex(0);                 // Вывод
   theQueue.insert(0);               // Вставка в конец очереди
   int v2;

   while( !theQueue.isEmpty() )     // Пока очередь не опустеет
      {
      int v1 = theQueue.remove();   // Извлечение вершины в начале очереди
                                    // Пока остаются непосещенные соседи
      while( (v2=getAdjUnvisitedVertex(v1)) != -1 )
         {                                  // Получение вершины
         vertexList[v2].wasVisited = true;  // Пометка
         displayVertex(v2);                 // Вывод
         theQueue.insert(v2);               // Вставка
         }
      }

   // Очередь пуста, обход закончен
   for(int j=0; j<nVerts; j++)            // Сброс флагов
      vertexList[j].wasVisited = false;
   }
```

Для графа, использованного ранее в программе dfs.java (см. рис. 13.8), метод bfs.java выводит следующий результат:

```
Visits: ABDCE
```

Программа bfs.java

Программа bfs.java, приведенная в листинге 13.2, имеет много общего с dfs.java; отличия невелики: вместо класса StackX в ней используется класс Queue (слегка измененный по сравнению с версией из главы 4), а метод dfs() заменен методом bfs().

Листинг 13.2. Программа bfs.java

```
// bfs.java
// Обход в ширину
// Запуск программы: C>java BFSApp
////////////////////////////////////////////////////////////////
class Queue
   {
   private final int SIZE = 20;
```

продолжение ⇨

Листинг 13.2 (*продолжение*)

```
   private int[] queArray;
   private int front;
   private int rear;
// -------------------------------------------------------------
   public Queue()            // Конструктор
      {
      queArray = new int[SIZE];
      front = 0;
      rear = -1;
      }
// -------------------------------------------------------------
   public void insert(int j) // Вставка элемента в конец очереди
      {
      if(rear == SIZE-1)
         rear = -1;
      queArray[++rear] = j;
      }
// -------------------------------------------------------------
public int remove()       // Извлечение элемента в начале очереди
   {
   int temp = queArray[front++];
   if(front == SIZE)
      front = 0;
   return temp;
}
// -------------------------------------------------------------
   public boolean isEmpty()  // true, если очередь пуста
      {
      return ( rear+1==front || (front+SIZE-1==rear) );
      }
// -------------------------------------------------------------
   }  // Конец класса Queue
////////////////////////////////////////////////////////////////
class Vertex
   {
   public char label;        // Метка (например, 'A')
   public boolean wasVisited;
// -------------------------------------------------------------
   public Vertex(char lab)   // Конструктор
      {
      label = lab;
      wasVisited = false;
      }
// -------------------------------------------------------------
   }  // Конец класса Vertex
////////////////////////////////////////////////////////////////
class Graph
   {
   private final int MAX_VERTS = 20;
```

```
   private Vertex vertexList[]; // Список вершин
   private int adjMat[][];      // Матрица смежности
   private int nVerts;          // Текущее количество вершин
   private Queue theQueue;
// ------------------
   public Graph()               // Конструктор
      {
      vertexList = new Vertex[MAX_VERTS];
                                // Матрица смежности
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // Матрица смежности
         for(int k=0; k<MAX_VERTS; k++)   // заполняется нулями
            adjMat[j][k] = 0;
      theQueue = new Queue();
      }
// -------------------------------------------------------------
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
      }
// -------------------------------------------------------------
   public void addEdge(int start, int end)
      {
      adjMat[start][end] = 1;
      adjMat[end][start] = 1;
      }
// -------------------------------------------------------------
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);
      }
// -------------------------------------------------------------
   public void bfs()                   // Обход в ширину
      {                                // Алгоритм начинает с вершины 0
      vertexList[0].wasVisited = true; // Пометка
      displayVertex(0);                // Вывод
      theQueue.insert(0);              // Вставка в конец очереди
      int v2;

      while( !theQueue.isEmpty() )     // Пока очередь не опустеет
         {
         int v1 = theQueue.remove();  // Извлечение вершины в начале очереди
                                       // Пока остаются непосещенные соседи

         while( (v2=getAdjUnvisitedVertex(v1)) != -1 )
            {                                 // Получение вершины
            vertexList[v2].wasVisited = true;  // Пометка
            displayVertex(v2);                 // Вывод
```

продолжение ⇨

Листинг 13.2 (*продолжение*)

```
            theQueue.insert(v2);               // Вставка
            }
         }

   // Очередь пуста, обход закончен
      for(int j=0; j<nVerts; j++)            // Сброс флагов
         vertexList[j].wasVisited = false;
      }
// ------------------------------------------------------------
   // Метод возвращает непосещенную вершину, смежную по отношению к v
   public int getAdjUnvisitedVertex(int v)
      {
      for(int j=0; j<nVerts; j++)
         if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
            return j;              // Возвращает первую найденную вершину
      return -1;                   // Таких вершин нет
      }
// ------------------------------------------------------------
   }  // Конец класса Graph
////////////////////////////////////////////////////////////////
class BFSApp
   {
   public static void main(String[] args)
      {
      Graph theGraph = new Graph();
      theGraph.addVertex('A');    // 0  (исходная вершина)
      theGraph.addVertex('B');    // 1
      theGraph.addVertex('C');    // 2
      theGraph.addVertex('D');    // 3
      theGraph.addVertex('E');    // 4
      theGraph.addEdge(0, 1);     // AB
      theGraph.addEdge(1, 2);     // BC
      theGraph.addEdge(0, 3);     // AD
      theGraph.addEdge(3, 4);     // DE

      System.out.print("Visits: ");
      theGraph.bfs();             // Обход в ширину
      System.out.println();
      }
   }
////////////////////////////////////////////////////////////////
```

Обход в ширину обладает одним интересным свойством: он сначала находит все вершины, находящиеся на расстоянии одного ребра от начальной вершины, затем все вершины на расстоянии двух ребер и т. д. Это свойство может пригодиться при поиске кратчайшего пути от начальной вершины к заданной. Запустите обход в ширину, и при обнаружении заданной вершины вы точно знаете, что построенный путь является кратчайшим путем к вершине. Если бы более короткий путь существовал, то алгоритм обхода в ширину нашел бы его ранее.

Минимальные остовные деревья

Представьте, что вы проектируете печатную плату (вроде изображенной на рис. 13.4), и хотите свести количество дорожек к минимуму. Иначе говоря, между контактами не должно быть лишних соединений, которые только занимают место и усложняют проводку других цепей.

Для решения этой задачи понадобится алгоритм, который бы для любого связного набора контактов и дорожек (вершин и ребер в терминологии графов) удалял все лишние дорожки. Результатом его работы является граф с минимальным количеством ребер, необходимых для соединения вершин. Например, граф на рис. 13.10, *а* состоит из пяти вершин с избыточными ребрами, а на рис. 13.10, *б* изображены те же пять вершин с минимальным количеством ребер, необходимым для их соединения. Эти ребра образуют *минимальное остовное дерево* (MST, Minimum Spanning Tree).

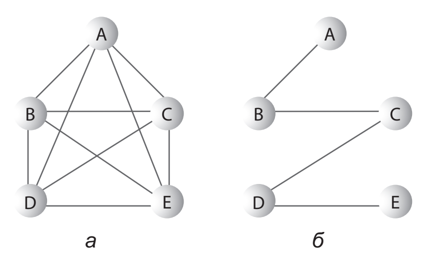


Рис. 13.10. Минимальное остовное дерево: *а* — избыточные ребра; *б* — минимальное количество ребер

Для заданного набора вершин существует много возможных минимальных остовных деревьев. На рис. 13.10, *б* обозначены ребра AB, BC, CD и DE, но с таким же успехом можно было составить минимальное остовное дерево из ребер AC, CE, ED и DB. Математически одаренные читатели наверняка заметят, что количество ребер E в минимальном остовном дереве всегда на единицу меньше количества вершин V:

$$E = V - 1.$$

Стоит напомнить, что длина ребер нас не интересует. Мы хотим определить не минимальную физическую длину, а минимальное количество ребер. (Ситуация изменится в следующей главе, когда речь пойдет о взвешенных графах.)

Алгоритм построения минимального остовного дерева почти идентичен алгоритму обхода. Он может базироваться как на обходе в глубину, так и на обходе в ширину. В нашем примере будет использоваться обход в глубину.

Это может показаться неожиданным, но обход в глубину с сохранением ребер, посещенных при поиске, приводит к автоматическому построению минимального остовного дерева. Единственное различие между методом построения минимального остовного дерева `mst()`, который будет приведен ниже, и методом обхода в глубину `dfs()`, приведенным ранее, заключается в том, что метод `mst()` должен каким-то образом сохранять посещенные ребра.

Приложение GraphN Workshop

Кнопка Tree в приложении GraphN Workshop строит минимальное остовное дерево для созданного вами графа. Опробуйте ее с разными графами. Вы увидите, что алгоритм выполняет те же действия, что и при обходе в глубину (кнопка DFS). Однако при использовании кнопки Tree ребра, включаемые в минимальное остовное дерево, темнеют. После завершения алгоритма приложение убирает все остальные ребра, а на графе остается только минимальное остовное дерево. Последнее нажатие кнопки Tree восстанавливает исходный граф на тот случай, если вы захотите снова воспользоваться им.

Реализация построения минимального остовного дерева на языке Java

Код метода mst() выглядит так:

```
while( !theStack.isEmpty() )       // Пока стек не опустеет
   {                               // Извлечение элемента из стека
   int currentVertex = theStack.peek();
   // get next unvisited neighbor
   int v = getAdjUnvisitedVertex(currentVertex);
   if(v == -1)                     // Если соседей больше нет,
      theStack.pop();              // извлечь элемент из стека
   else                            // Сосед существует
      {
      vertexList[v].wasVisited = true;  // Пометка
      theStack.push(v);                 // Занесение в стек
                                        // Вывод ребра
      displayVertex(currentVertex);     // От currentVertex
      displayVertex(v);                 // к v
      System.out.print(" ");
      }
   }

   // Стек пуст, работа закончена
   for(int j=0; j<nVerts; j++)          // Сброс флагов
      vertexList[j].wasVisited = false;
}
```

Как видите, код метода очень похож на код dfs(). В секции else выводится текущая вершина и следующий непосещенный сосед. Эти две вершины определяют ребро, по которому алгоритм перемещается для получения новой вершины. Такие ребра составляют минимальное остовное дерево.

В методе main() программы mst.java фрагмент построения графа выглядит так:

```
Graph theGraph = new Graph();
theGraph.addVertex('A');    // 0  (исходная вершина)
theGraph.addVertex('B');    // 1
theGraph.addVertex('C');    // 2
```

```
theGraph.addVertex('D');    // 3
theGraph.addVertex('E');    // 4

theGraph.addEdge(0, 1);     // AB
theGraph.addEdge(0, 2);     // AC
theGraph.addEdge(0, 3);     // AD
theGraph.addEdge(0, 4);     // AE
theGraph.addEdge(1, 2);     // BC
theGraph.addEdge(1, 3);     // BD
theGraph.addEdge(1, 4);     // BE
theGraph.addEdge(2, 3);     // CD
theGraph.addEdge(2, 4);     // CE
theGraph.addEdge(3, 4);     // DE
```

Построенный граф показан на рис. 13.10, *а*. Когда метод `mst()` завершит свою работу, в графе остаются только четыре ребра, показанных на рис. 13.10, *б*. Результат выполнения программы `mst.java` выглядит так:

```
Minimum spanning tree: AB BC CD DE
```

Как уже говорилось выше, это лишь одно из многих возможных минимальных остовных деревьев, которые могут быть построены для приведенного графа. Например, если начать с другой вершины, алгоритм построит другое дерево. К таким же последствиям приведут небольшие вариации в коде — скажем, если обход в методе `getAdjUnvisitedVertex()` будет начинаться не с начала, а с конца списка `vertexList[]`.

Минимальное остовное дерево естественным образом строится на базе обхода в глубину, потому что этот алгоритм посещает все узлы, но только по одному разу. Он никогда не переходит к узлу, который уже был посещен ранее, а все его перемещения по ребрам ограничены строгой необходимостью. Таким образом, путь алгоритма по графу должен образовывать минимальное остовное дерево.

Программа mst.java

В листинге 13.3 приведена программа `mst.java`. Она отличается от программы `dfs.java` только методом `mst()` и построением графа в `main()`.

Листинг 13.3. Программа mst.java

```
// mst.java
// Построение минимального остовного дерева
// Запуск программы: C>java MSTApp
////////////////////////////////////////////////////////////////
class StackX
   {
   private final int SIZE = 20;
   private int[] st;
   private int top;
// ------------------------------------------------------------
   public StackX()           // Конструктор
      {
```

продолжение ⇗

Листинг 13.3 (*продолжение*)

```
      st = new int[SIZE];    // Создание массива
      top = -1;
      }
// -------------------------------------------------------------
   public void push(int j)   // Размещение элемента в стеке
      { st[++top] = j; }
// -------------------------------------------------------------
   public int pop()          // Извлечение элемента из стека
      { return st[top--]; }
// -------------------------------------------------------------
   public int peek()         // Чтение с вершины стека
      { return st[top]; }
// -------------------------------------------------------------
   public boolean isEmpty()  // true, если стек пуст
      { return (top == -1); }
// -------------------------------------------------------------
   }  // Конец класса StackX
////////////////////////////////////////////////////////////////
class Vertex
   {
   public char label;        // Метка (например, 'A')
   public boolean wasVisited;
// -------------------------------------------------------------
   public Vertex(char lab)   // Конструктор
      {
      label = lab;
      wasVisited = false;
      }
// -------------------------------------------------------------
   }  // Конец класса Vertex
////////////////////////////////////////////////////////////////
class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // Список вершин
   private int adjMat[][];      // Матрица смежности
   private int nVerts;          // Текущее количество вершин
   private StackX theStack;
// -------------------------------------------------------------
   public Graph()               // Конструктор
      {
      vertexList = new Vertex[MAX_VERTS];
                                // Матрица смежности
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // Матрица смежности
         for(int k=0; k<MAX_VERTS; k++)   // заполняется нулями
            adjMat[j][k] = 0;
```

```
      theStack = new StackX();
      }
// -------------------------------------------------------------
   public void addEdge(int start, int end)
      {
      adjMat[start][end] = 1;
      adjMat[end][start] = 1;
      }
// -------------------------------------------------------------
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);
      }
// -------------------------------------------------------------
   public void mst()  // Построение минимального остовного дерева
      {                                // 
      vertexList[0].wasVisited = true;   // Пометка
      theStack.push(0);                  // Занесение в стек
      while( !theStack.isEmpty() )       // Пока стек не опустеет
         {                               // Извлечение элемента из стека
         int currentVertex = theStack.peek();
                                         // Получение следующего соседа
         int v = getAdjUnvisitedVertex(currentVertex);
         if(v == -1)                     // Если соседей больше нет,
            theStack.pop();              // извлечь элемент из стека
         else                            // Сосед существует
            {
            vertexList[v].wasVisited = true;  // Пометка
            theStack.push(v);                 // Занесение в стек
                                              // Вывод ребра
            displayVertex(currentVertex);     // От currentVertex
            displayVertex(v);                 // к v
            System.out.print(" ");
            }
         }

         // Стек пуст, работа закончена
         for(int j=0; j<nVerts; j++)          // Сброс флагов
            vertexList[j].wasVisited = false;
      }
// -------------------------------------------------------------
   // Метод возвращает непосещенную вершину, смежную по отношению к v
   public int getAdjUnvisitedVertex(int v)
      {
      for(int j=0; j<nVerts; j++)
         if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
            return j;
      return -1;
   }
```

продолжение ⇗

Листинг 13.3 (*продолжение*)

```
// -------------------------------------------------------------
   }  // Конец класса Graph
////////////////////////////////////////////////////////////////
class MSTApp
   {
   public static void main(String[] args)
      {
      Graph theGraph = new Graph();
      theGraph.addVertex('A');    // 0  (исходная вершина)
      theGraph.addVertex('B');    // 1
      theGraph.addVertex('C');    // 2
      theGraph.addVertex('D');    // 3
      theGraph.addVertex('E');    // 4
      theGraph.addEdge(0, 1);     // AB
      theGraph.addEdge(0, 2);     // AC
      theGraph.addEdge(0, 3);     // AD
      theGraph.addEdge(0, 4);     // AE
      theGraph.addEdge(1, 2);     // BC
      theGraph.addEdge(1, 3);     // BD
      theGraph.addEdge(1, 4);     // BE
      theGraph.addEdge(2, 3);     // CD
      theGraph.addEdge(2, 4);     // CE
      theGraph.addEdge(3, 4);     // DE

      System.out.print("Minimum spanning tree: ");
      theGraph.mst();             // Минимальное остовное дерево
      System.out.println();
      }
   }  // Конец класса MSTApp
////////////////////////////////////////////////////////////////
```

Граф, построенный в `main()`, представляет собой пятиконечную звезду, в которой каждый узел соединен со всеми остальными узлами. Программа выводит следующий результат:

```
Minimum spanning tree: AB BC CD DE
```

Топологическая сортировка с направленными графами

Топологическая сортировка также принадлежит к числу операций, которые могут моделироваться при помощи графов. Она удобна в ситуациях, в которых требуется расставить некоторые элементы или события в определенном порядке. Рассмотрим пример.

Пример

В институтах и колледжах студент не может выбрать любой учебный курс по своему усмотрению. У некоторых курсов имеются необходимые условия — обязательное прохождение других учебных курсов. С другой стороны, некоторые курсы могут быть необходимым условием для получения ученой степени в определенной области. На рис. 13.11 изображена (условная) структура курсов, необходимых для получения ученой степени по математике.

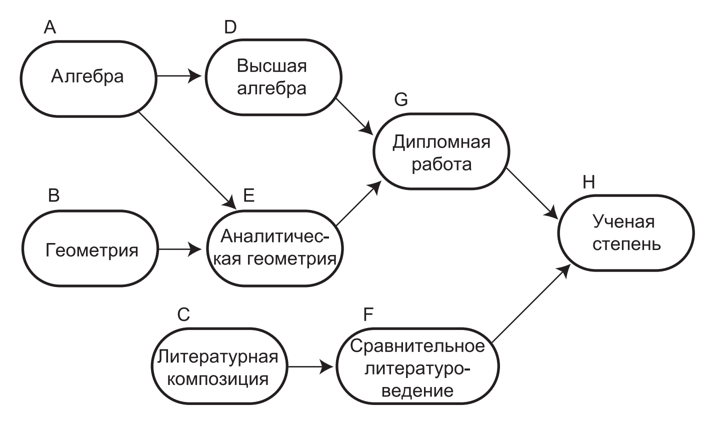


Рис. 13.11. Обязательные курсы

Чтобы получить ученую степень, необходимо сдать дипломную работу и (из-за давления со стороны факультета английского языка) сравнительное литературоведение. Однако сдача дипломной работы возможна лишь при условии сдачи высшей алгебры и аналитической геометрии, а курс сравнительного литературоведения доступен лишь после прохождения курса литературной композиции. Кроме того, геометрия является обязательным условием для курса аналитической геометрии, а алгебра — для высшей алгебры и аналитической геометрии.

Направленные графы

Как видно из рис. 13.11, для представления подобной структуры курсов можно воспользоваться графом. Однако граф этот должен обладать свойством, с которым мы еще не сталкивались: каждое его ребро должно обладать направлением. Такие графы называются *направленными*. В направленном графе переходы по ребрам возможны только в одном из двух направлений. На схеме графа такие направления обозначаются стрелками.

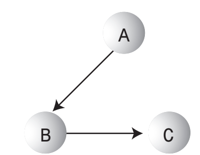


Рис. 13.12. Направленный граф

В программе основное различие между ненаправленным и направленным графом заключается в том, что ребро направленного графа обозначается только одним элементом матрицы смежности. На рис. 13.12

изображен небольшой направленный граф, а в табл. 13.5 приведена его матрица смежности.

Таблица 13.5. Матрица смежности направленного графа

	A	B	C
A	0	1	0
B	0	0	1
C	0	0	0

Каждое ребро представлено одним значением 1. Заголовки строк определяют начало направленного ребра, а заголовки столбцов — его конец. Таким образом, ребро от вершины A в B представлено значением 1 на пересечении строки A и столбца B. Если бы направление ребра изменилось так, что оно будет вести от B в A, то «единица» появится на пересечении строки B и столбца A.

Для ненаправленных графов, как упоминалось ранее, половина матрицы смежности является симметричным отражением другой половины, а половина ячеек оказывается лишней. Но для взвешенного графа каждая ячейка матрицы смежности несет полезную информацию. Две треугольные части матрицы не симметричны.

В направленном графе метод для добавления ребра состоит всего из одной команды:

```
public void addEdge(int start, int end)  // Направленный граф
   {
   adjMat[start][end] = 1;
   }
```

вместо двух команд, необходимых для ненаправленного графа.

Если же для представления графа используется матрица смежности, то в списке A присутствует вершина B, но (в отличие от ненаправленного графа) в списке B вершины A нет.

Топологическая сортировка

Допустим, вы составляете список всех курсов, необходимых для получения ученой степени, на основе рис. 13.11. Курсы необходимо выстроить в порядке их прохождения. Ученая степень находится на последней позиции этого списка, который может выглядеть так:

```
BAEDGCFH
```

Граф, упорядоченный подобным образом, называется *топологически отсортированным*. Каждый учебный курс, который должен быть пройден ранее некоторого другого курса, предшествует ему в списке.

Вообще говоря, существует много разных конфигураций, соответствующих требованиям порядка прохождения курсов. Например, можно начать с литературных курсов C и F:

```
CFBAEDGH
```

Эта последовательность тоже выполняет все обязательные условия. Также существует много других возможных решений. Результаты, генерируемые алгоритмом построения топологических сортировок, зависят от выбранного метода и технических подробностей кода.

Топологическая сортировка также применяется для моделирования других ситуаций, например при планировании операций. Скажем, при сборке машины тормоза должны устанавливаться раньше колес, а двигатель должен быть собран до его закрепления на раме. Фирмы-производители автомобилей используют графы для моделирования тысяч операций производственного процесса, чтобы обеспечить их выполнение в правильном порядке.

Моделирование планирования заданий с использованием графов называется *анализом критических путей*. Применение взвешенного графа (см. следующую главу) позволяет включить в граф информацию о времени, необходимом для выполнения различных задач проекта. Например, при помощи такого графа можно вычислить минимальное время, необходимое для выполнения всего проекта.

Приложение GraphD Workshop

Приложение GraphD Workshop моделирует направленные графы. Оно работает почти так же, как GraphN, но рядом с каждым ребром выводится точка, обозначающая его направление. Будьте внимательны: направление ребра определяется направлением, в котором перетаскивается указатель мыши при его создании. На рис. 13.13 изображено приложение GraphD Workshop, использованное для моделирования графа учебных курсов на рис. 13.11.

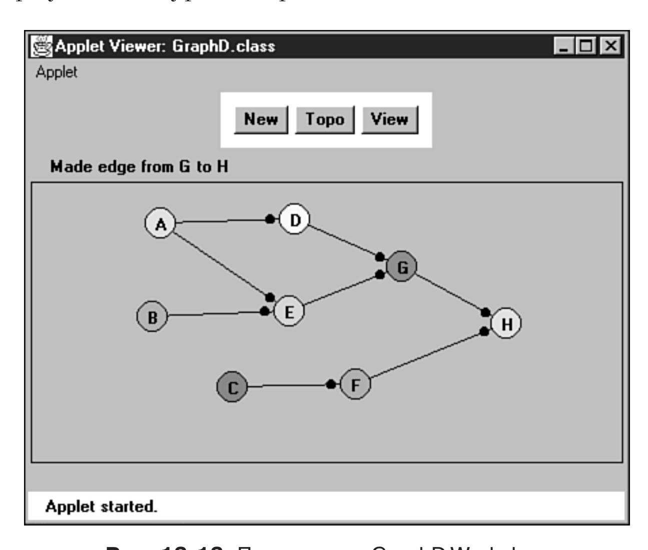


Рис. 13.13. Приложение GraphD Workshop

Алгоритм топологической сортировки основан на нестандартной, но простой идее. Он состоит из двух шагов.

ШАГ 1

Найти вершину, не имеющую преемников.

Преемниками вершины называются вершины, расположенные непосредственно «за ней», то есть соединенные с ней ребром, указывающим в их направлении. Если в графе существует ребро, ведущее от A к B, то B является преемником A. На рис. 13.11 преемники существуют у всех вершин, кроме H.

ШАГ 2

Удалить эту вершину из графа и вставить ее метку в начало списка.

Шаги 1 и 2 повторяются до тех пор, пока из графа не будут удалены все вершины. На этой стадии список состоит из всех вершин, упорядоченных в топологическом порядке.

За тем, как работает этот процесс, можно понаблюдать в приложении GraphD. Постройте граф на рис. 13.11 (или любой другой граф на ваше усмотрение); вершины создаются двойным щелчком, а ребра — перетаскиванием. Далее многократно нажимайте на кнопку `Topo`. С удалением очередной вершины из графа ее метка размещается в списке под графом.

Удаление вершины может показаться слишком радикальным действием, но оно занимает центральное место в алгоритме. Алгоритм не может найти вторую удаляемую вершину, пока первая вершина не будет удалена. При необходимости данные графа (список вершин и матрицу смежности) можно сохранить во временных переменных и восстановить после завершения сортировки, как это делается в приложении GraphD.

Работа алгоритма основана на том факте, что вершина, не имеющая преемников, должна быть последней в топологическом порядке. Сразу же после ее удаления появляется другая вершина, не имеющая преемников; она должна стать предпоследней в результатах сортировки и т. д.

Алгоритм топологической сортировки работает как со связными, так и с несвязными графами. Несвязные графы моделируют ситуацию с наличием несвязанных целей, например получением ученой степени по математике с одновременным получением медицинского сертификата по оказанию экстренной помощи.

Циклы и деревья

Одну из разновидностей графов, с которыми не может справиться алгоритм топологической сортировки, составляют графы с циклами. Что такое цикл? Это путь, который завершается в той точке, с которой он начинается. На рис. 13.14 путь B-C-D-B образует цикл. (Обратите внимание: путь A-B-C-A циклом не является, потому что переход от C к A невозможен.)

Цикл моделирует «порочный круг», при котором прохождение курса B является обязательным условием для курса C, прохождение курса C — для курса D, а курс D обязателен для B.

Граф, не содержащий циклов, называется *деревом*. Двоичные и многопутевые деревья, представленные ранее в этой книге, являются деревьями в этом смысле. Однако понятие дерева в теории графов более универсально по сравнению с двоичными и многопутевыми деревьями, имеющими фиксированное количество потомков. В графе вершина дерева может соединяться с произвольным количеством других вершин (при условии, что при этом не образуются циклы).

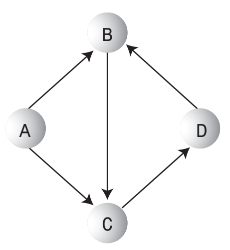


Рис. 13.14. Граф с циклом

Наличие циклов в ненаправленном графе проверяется достаточно легко. Если граф с N узлами содержит более $N - 1$ ребер, в нем заведомо существуют циклы. Чтобы убедиться в этом, попробуйте нарисовать граф из N узлов и N ребер, не имеющий циклов.

Топологическая сортировка должна применяться к направленным графам, не содержащим циклов. Такие графы называются *направленными ациклическими графами*.

Реализация на языке Java

Реализация метода topo(), выполняющего топологическую сортировку, на языке Java выглядит так:

```
public void topo()           // Топологическая сортировка
   {
   int orig_nVerts = nVerts;  // Сохранение количества вершин

   while(nVerts > 0)          // Пока в графе остаются вершины
   {
      // Получение вершины без преемников или -1
      int currentVertex = noSuccessors();
      if(currentVertex == -1)       // В графе есть цикл
         {
         System.out.println("ERROR: Graph has cycles");
         return;
         }
      // Вставка метки вершины в массив (начиная с конца)
      sortedArray[nVerts-1] = vertexList[currentVertex].label;

      deleteVertex(currentVertex);  // Удаление вершины
      }
   // Все вершины удалены, вывод содержимого sortedArray
   System.out.print("Topologically sorted order: ");
   for(int j=0; j<orig_nVerts; j++)
      System.out.print( sortedArray[j] );
   System.out.println("");
   }
```

Вся работа выполняется в цикле while, который продолжает выполняться до тех пор, пока количество вершин не сократится до 0. Метод работает по следующему алгоритму:

1. Вызов noSuccessors() ищет любую вершину, не имеющую преемников.
2. Если вершина найдена, ее метка помещается в конец массива sortedArray[], а сама вершина удаляется из графа.
3. Если вершина не найдена, значит, в графе существует цикл.

Последняя удаляемая вершина должна находиться на первом месте в списке, поэтому метки вершин помещаются в sortedArray, начиная с конца, и продвигаются к началу с уменьшением значения nVerts (количества вершин в графе).

Если в графе остаются вершины, но у всех вершин имеются преемники, значит, граф содержит цикл, а алгоритм выводит сообщение и завершает работу. При отсутствии циклов происходит выход из цикла while, а список из sortedArray выводится в порядке топологической сортировки вершин.

Метод noSuccessors() использует матрицу смежности для поиска вершин, не имеющих преемников. Внешний цикл for перебирает строки, последовательно просматривая все вершины. Для каждой вершины во внутреннем цикле for перебираются столбцы в поисках значения 1. Если такой столбец будет найден, значит, у вершины имеется преемник (существует ребро, ведущее от текущей вершины к другой). При обнаружении 1 происходит выход из внутреннего цикла для проверки следующей вершины.

Только если будет найдена строка, в которой нет ни одной 1, алгоритм знает, что он обнаружил вершину без преемников; в таком случае возвращается номер этой строки. Если вершина не обнаружена, возвращается –1. Код метода noSuccessors():

```
public int noSuccessors()  // Метод возвращает вершину, не имеющую
   {                       // преемников (или -1, если ее нет)
   boolean isEdge;  // Ребро в матрице adjMat, ведущее от row в column

   for(int row=0; row<nVerts; row++)  // Для каждой вершины
      {
      isEdge = false;                 // Проверка ребер
      for(int col=0; col<nVerts; col++)
         {
         if( adjMat[row][col] > 0 )   // Если существует ребро
            {                         // от текущей вершины
            isEdge = true;
            break;                    // У вершины имеется преемник
            }                         // Переход к проверке
         }                            // другой вершины
      if( !isEdge )                   // Если ребер нет,
         return row;                  // то нет и преемников
      }
   return -1;                         // Вершина не найдена
   }
```

Удаление вершины выполняется просто, не считая некоторых подробностей. Вершина удаляется из массива vertexList[], а находящиеся выше нее вершины

сдвигаются вниз, заполняя свободное место. Аналогичным образом строка и столбец вершины удаляются из матрицы смежности, а строки и столбцы выше и справа от них сдвигаются вниз и влево для заполнения свободных мест. Эти операции выполняются методами deleteVertex(), moveRowUp() и moveColLeft(), с которыми можно ознакомиться в полном коде topo.java (листинг 13.4). Вообще говоря, для этого алгоритма более эффективным было бы представление графа в формате списков смежности, но его реализация отвлечет нас от основной темы.

Метод main() строит граф на рис. 13.10 вызовами методов, аналогичными тем, которые приводились ранее. Метод addEdge(), как уже было сказано, вставляет в матрицу смежности только одно число, потому что граф является направленным.

Код метода main():

```
public static void main(String[] args)
   {
   Graph theGraph = new Graph();
   theGraph.addVertex('A');    // 0
   theGraph.addVertex('B');    // 1
   theGraph.addVertex('C');    // 2
   theGraph.addVertex('D');    // 3
   theGraph.addVertex('E');    // 4
   theGraph.addVertex('F');    // 5
   theGraph.addVertex('G');    // 6
   theGraph.addVertex('H');    // 7
   theGraph.addEdge(0, 3);     // AD
   theGraph.addEdge(0, 4);     // AE
   theGraph.addEdge(1, 4);     // BE
   theGraph.addEdge(2, 5);     // CF
   theGraph.addEdge(3, 6);     // DG
   theGraph.addEdge(4, 6);     // EG
   theGraph.addEdge(5, 7);     // FH
   theGraph.addEdge(6, 7);     // GH
   theGraph.topo();            // Сортировка
   }
```

После того как граф будет построен, main() сортирует его вызовом topo() и выводит результат. Выходные данные выглядят так:

```
Topologically sorted order: BAEDGCFH
```

Конечно, метод main() можно изменить, чтобы он строил другие графы.

Полный код программы topo.java

Мы уже рассмотрели большинство методов topo.java. В листинге 13.4 приведен полный код.

Листинг 13.4. Программа topo.java

```
// topo.java
// Топологическая сортировка
// Запуск программы: C>java TopoApp
```

продолжение ⇗

Листинг 13.4 (*продолжение*)

```
////////////////////////////////////////////////////////////////
class Vertex
   {
   public char label;        // Метка (например, 'A')
// -------------------------------------------------------------
   public Vertex(char lab)   // Конструктор
      { label = lab; }
   } Конец класса Vertex
////////////////////////////////////////////////////////////////
class Graph
   {
   private final int MAX_VERTS = 20;
   private Vertex vertexList[]; // Список вершин
   private int adjMat[][];      // Матрица смежности
   private int nVerts;          // Текущее количество вершин
   private char sortedArray[];
// -------------------------------------------------------------
   public Graph()               // Конструктор
      {
      vertexList = new Vertex[MAX_VERTS];
                                // Матрица смежности
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // Матрица смежности
         for(int k=0; k<MAX_VERTS; k++)   // заполняется нулями
            adjMat[j][k] = 0;
      sortedArray = new char[MAX_VERTS];  // Метки отсортированных вершин
   }
// -------------------------------------------------------------
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
      }
// -------------------------------------------------------------
   public void addEdge(int start, int end)
      {
      adjMat[start][end] = 1;
      }
// -------------------------------------------------------------
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);
      }
// -------------------------------------------------------------
   public void topo()  // Топологическая сортировка
      {
      int orig_nVerts = nVerts;  // Сохранение количества вершин

      while(nVerts > 0)  // Пока в графе остаются вершины
```

```
         {
         // Получение вершины без преемников или -1
         int currentVertex = noSuccessors();
         if(currentVertex == -1)       // В графе есть цикл
            {
            System.out.println("ERROR: Graph has cycles");
            return;
            }
         // Вставка метки вершины в массив (начиная с конца)
         sortedArray[nVerts-1] = vertexList[currentVertex].label;

         deleteVertex(currentVertex);  // Удаление вершины
         }

      // Все вершины удалены, вывод содержимого sortedArray
      System.out.print("Topologically sorted order: ");
      for(int j=0; j<orig_nVerts; j++)
         System.out.print( sortedArray[j] );
      System.out.println("");
      }
// -------------------------------------------------------------
   public int noSuccessors()  // Метод возвращает вершину, не имеющую
      {                       // преемников (или -1, если ее нет)
      boolean isEdge;  // Ребро в матрице adjMat, ведущее от row в column

      for(int row=0; row<nVerts; row++)  // Для каждой вершины
         {
         isEdge = false;                 // Проверка ребер
         for(int col=0; col<nVerts; col++)
            {
            if( adjMat[row][col] > 0 )   // Если существует ребро
               {                         // от текущей вершины
               isEdge = true;
               break;                    // У вершины имеется преемник
               }                         // Переход к проверке
            }                            // другой вершины
         if( !isEdge )                   // Если ребер нет,
            return row;                  // то нет и преемников
         }
      return -1;                         // Вершина не найдена
      }
// -------------------------------------------------------------
   public void deleteVertex(int delVert)
      {
      if(delVert != nVerts-1)      // Если вершина не последняя
         {                         // Удаление из vertexList
         for(int j=delVert; j<nVerts-1; j++)
            vertexList[j] = vertexList[j+1];
                                   // удаление строки из adjMat
         for(int row=delVert; row<nVerts-1; row++)
```

продолжение ⇨

Листинг 13.4 (*продолжение*)

```
         moveRowUp(row, nVerts);
                                 // Удаление столбца из adjMat
      for(int col=delVert; col<nVerts-1; col++)
         moveColLeft(col, nVerts-1);
      }
   nVerts--;                     // Количество вершин уменьшается на 1
   }
// -------------------------------------------------------------
   private void moveRowUp(int row, int length)
   {
   for(int col=0; col<length; col++)
      adjMat[row][col] = adjMat[row+1][col];
   }
// -------------------------------------------------------------
   private void moveColLeft(int col, int length)
   {
   for(int row=0; row<length; row++)
      adjMat[row][col] = adjMat[row][col+1];
   }
// -------------------------------------------------------------
   }  // Конец класса Graph
////////////////////////////////////////////////////////////////
class TopoApp
   {
   public static void main(String[] args)
      {
      Graph theGraph = new Graph();
      theGraph.addVertex('A');    // 0
      theGraph.addVertex('B');    // 1
      theGraph.addVertex('C');    // 2
      theGraph.addVertex('D');    // 3
      theGraph.addVertex('E');    // 4
      theGraph.addVertex('F');    // 5
      theGraph.addVertex('G');    // 6
      theGraph.addVertex('H');    // 7
      theGraph.addEdge(0, 3);     // AD
      theGraph.addEdge(0, 4);     // AE
      theGraph.addEdge(1, 4);     // BE
      theGraph.addEdge(2, 5);     // CF
      theGraph.addEdge(3, 6);     // DG
      theGraph.addEdge(4, 6);     // EG
      theGraph.addEdge(5, 7);     // FH
      theGraph.addEdge(6, 7);     // GH
      theGraph.topo();            // Сортировка
      }
   }  // Конец класса TopoApp
////////////////////////////////////////////////////////////////
```

В следующей главе будут рассмотрены графы, в которых ребрам присваиваются не только направления, но и веса.

Связность в направленных графах

Вы уже знаете, как в ненаправленном графе выполняется поиск всех соединенных узлов (обход в глубину или в ширину). При поиске всех соединенных вершин в направленном графе ситуация усложняется. Алгоритм уже не может начать со случайно выбранной вершины и рассчитывать на то, что ему удастся достичь всех остальных соединенных с ней вершин.

Для примера рассмотрим граф на рис. 13.15. Если начать с вершины A, то алгоритм сможет добраться до вершины C, но не до других вершин. Начиная с B, он не сможет добраться до вершины D, а от вершины C он вообще никуда не сможет попасть. В том, что касается связности, необходимо найти ответ на один важный вопрос: каких вершин можно достичь, если начать обход с конкретной вершины?

Таблица связности

Программу `dfs.java` (см. листинг 13.1) легко изменить таким образом, чтобы обход последовательно начинался с каждой вершины. Вот как выглядит результат ее выполнения для графа на рис. 13.15:

```
AC
BACE
C
DEC
EC
```

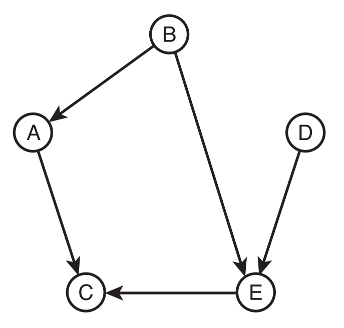


Рис. 13.15. Направленный граф

Мы получили *таблицу связности* для направленного графа. Первая буква в строке определяет начальную вершину, а остальные буквы — вершины, к которым можно перейти (напрямую или через другие вершины) от начальной вершины.

Алгоритм Уоршелла

В некоторых приложениях необходимо быстро определить, возможен ли переход от одной вершины к другой. Допустим, вы хотите перелететь из Афин в Мурманск, причем количество промежуточных пересадок вас не беспокоит. Возможен ли такой перелет?

Можно проанализировать таблицу связности, но это потребует просмотра всех элементов отдельной строки за время $O(N)$ (где N — среднее количество вершин, доступных от заданной вершины). У вас нет лишнего времени; существует ли более быстрый способ?

Оказывается, для графа можно построить таблицу, которая мгновенно (то есть за время $O(1)$) сообщит вам, возможен ли переход от одной вершины к другой. Такая таблица строится систематическим изменением матрицы смежности графа. Граф, представленный такой видоизмененной матрицей смежности, называется *транзитивным замыканием* исходного графа.

Напомним, что в обычной матрице смежности номер строки определяет начало ребра, а номер столбца — его конец. (Таблица связности имеет такую же структуру.) Единица на пересечении строки C и столбца D означает, что в графе существует ребро, ведущее от вершины C к вершине D. От одной вершины к другой можно перейти за один шаг. (Конечно, в направленном графе из этого не следует существование обратного перехода, из D в C.) В таблице 13.6 приведена матрица смежности для графа на рис. 13.15.

Таблица 13.6. Матрица смежности

	A	**B**	**C**	**D**	**E**
A	0	0	1	0	0
B	1	0	0	0	1
C	0	0	0	0	0
D	0	0	0	0	1
E	0	0	1	0	0

Преобразование матрицы смежности в транзитивное замыкание графа осуществляется по *алгоритму Уоршелла*. Этот алгоритм, выполняющий довольно значительную работу в нескольких строках кода, основан на простой идее:

> Если от вершины L можно перейти к вершине M, а от вершины M — к вершине N, то можно перейти и от вершины L к вершине N.

Двухшаговый путь создается из двух одношаговых путей. Матрица смежности содержит все допустимые одношаговые пути, поэтому она станет хорошей отправной точкой для применения этого правила.

Найдет ли алгоритм пути длиной более двух ребер? В конце концов, в правиле речь идет об объединении двух одношаговых путей в один двухшаговый. Оказывается, алгоритм способен строить пути произвольной длины на основании найденных ранее многошаговых путей. Приведенная ниже реализация гарантирует результат, а его теоретическое обоснование выходит за рамки книги.

Итак, мы рассмотрим работу алгоритма на примере табл. 13.6. Мы последовательно, строку за строкой проанализируем каждую ячейку матрицы смежности.

Строка A

Начнем со строки A. В столбцах A и B стоят нули, но столбец C содержит 1; остановимся на нем.

Единица в этой ячейке означает, что в графе существует путь от A к C. Если бы нам было известно, что также существует путь от другой вершины X к A, то можно было бы утверждать и о существовании пути от X к C.

Какие ребра графа заканчиваются в вершине A (и есть ли такие ребра?)? Их следует искать в столбце A. Из табл. 13.6 видно, что столбец A содержит всего одну единицу в строке B. Она означает, что в графе существует ребро от вершины B к A.

Таким образом, существует ребро от B к A и (исходное) ребро от A к C. Из этого следует, что от B к C можно перейти за два шага. В этом можно убедиться по рис. 13.15. Чтобы сохранить этот результат, мы ставим 1 на пересечении строки B и столбца C (рис. 13.16, *а*).

Остальные ячейки строки A пока остаются пустыми.

Строки B, C и D

Переходим к строке B. Первая ячейка (столбец A) содержит 1, указывая на существование ребра от B к A. Существуют ли другие ребра, заканчивающиеся в B? Столбец B пуст; следовательно, ни одна из единиц в строке B не приведет к обнаружению более длинного пути, потому что ни одно ребро не заканчивается в B.

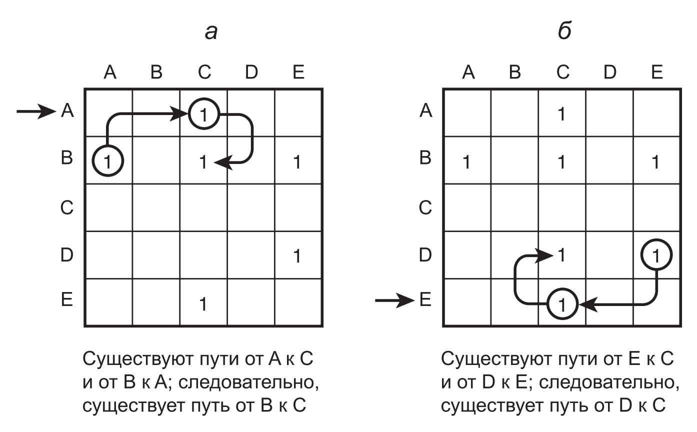


Рис. 13.16. Последовательность действий в алгоритме Уоршелла: *а*) *y* = 0; *б*) *y* = 4

Строка C не содержит ни одной единицы, переходим к строке D. Здесь находится ребро от D к E. Однако столбец D пуст; соответственно ребер, заканчивающихся на вершине D, не существует.

Строка E

В строке E отыскивается ребро от E к C. Просматривая столбец E, мы находим в нем ребро от B к E; из существования путей от B к E и от E к C делается вывод о существовании пути от B к C. Однако этот путь уже был обнаружен ранее, на что указывает 1 в ячейке таблицы. Пути от D к E и от E к C образуют путь от D к C, в эту ячейку ставится 1. Результат показан на рис. 13.16, *б*.

На этом работа алгоритма Уоршелла завершается. В матрицу смежности были добавлены две единицы; измененная матрица показывает, к каким узлам можно перейти от другого узла за любое количество шагов. Если нарисовать граф на основе новой матрицы, получится транзитивное замыкание исходного графа на рис. 13.15.

Реализация алгоритма Уоршелла

Один из способов реализации алгоритма Уоршелла основан на использовании трех вложенных циклов (предложен Седжвиком — см. приложение Б). Внешний цикл перебирает строки (переменная y). Средний цикл перебирает все ячейки строки (переменная x). Если в ячейке (x,y) обнаруживается 1, значит, в графе существует ребро от y к x; активизируется третий (внутренний) цикл с переменной z.

Третий цикл просматривает ячейки в столбце y и ищет ребро, завершающееся в y. (Обратите внимание: в первом цикле переменная y используется для перебора строк, а в третьем цикле она индексирует столбцы.) Если элемент на пересечении столбца y со строкой z содержит 1, значит, существует ребро от z к y. Из факта существования двух ребер — от z к y и от y к x — делается вывод о существовании пути от z к x, вследствие чего 1 размещается в ячейке (x, z). Подробности реализации остаются читателю для самостоятельной работы.

Итоги

- Графы состоят из вершин, соединенных ребрами.
- Графы могут представлять многие реально существующие сущности: пути авиаперелетов, электрические цепи, планируемые задачи и т. д.
- Алгоритмы обхода решают задачу систематического посещения каждой вершины графа. Обход лежит в основе ряда других операций.
- Два основных алгоритма обхода — обход в глубину и обход в ширину.
- Алгоритм обхода в глубину обычно реализуется на базе стека, а алгоритм обхода в ширину — на базе очереди.
- Минимальное остовное дерево состоит из минимального количества ребер, необходимых для соединения всех вершин графа.
- Для построения минимального остовного дерева используется незначительная модификация алгоритма обхода в глубину для невзвешенного графа.
- В направленном графе ребра обладают направлением (часто обозначаемым стрелкой).
- Алгоритм топологической сортировки создает список вершин, упорядоченных таким образом, что при наличии пути от A к B вершина A предшествует B в списке.
- Топологическая сортировка может выполняться только с направленными ациклическими графами.
- Топологическая сортировка обычно используется при планировании сложных проектов, состоящих из задач, выполнение которых возможно только после завершения других задач.
- Алгоритм Уоршелла определяет возможность перехода (прямого или многошагового) от произвольной вершины к любой другой вершине.

Вопросы

Следующие вопросы помогут читателю проверить качество усвоения материала. Ответы на них приведены в приложении В.

1. В графе ________ соединяет две __________ .
2. Как по матрице смежности определить количество ребер в ненаправленном графе?
3. Какая структурная единица графа соответствует выбору хода в игровом моделировании?
4. Направленным называется граф, в котором:
   a) ребра образуют минимальное остовное дерево;
   b) из вершины A возможен переход только в вершину B, затем в вершину C и т. д.;
   c) от одной вершины к другой можно перейти только в одном направлении;
   d) по любому заданному пути можно перемещаться только в одном направлении.
5. Матрица смежности состоит из строк {0,1,0,0}, {1,0,1,1}, {0,1,0,0} и {0,1,0,0}. Как выглядит соответствующий список смежности?
6. Минимальным остовным деревом называется граф, в котором:
   a) количество ребер, соединяющих все вершины, минимально;
   b) количество ребер равно количеству вершин;
   c) все избыточные вершины были удалены;
   d) каждая пара вершин соединяется минимальным количеством ребер.
7. Сколько разных минимальных основных деревьев существует в ненаправленном графе из трех вершин и трех ребер?
8. Ненаправленный граф обязательно содержит цикл, если:
   a) к любой вершине можно перейти от другой вершины;
   b) количество путей больше количества вершин;
   c) количество ребер равно количеству вершин;
   d) количество путей меньше количества ребер.
9. Граф, не содержащий циклов, называется _______.
10. Может ли минимальное остовное дерево ненаправленного графа содержать циклы?
11. Для заданного графа может существовать несколько правильных вариантов топологической сортировки (Да/Нет).
12. Результатом топологической сортировки является:
    a) упорядочение вершин, при котором все направленные ребра идут в одном направлении;
    b) упорядочение вершин в порядке возрастания количества ребер;

c) упорядочение вершин, при котором A предшествует B, B предшествует C и т. д.;

d) упорядочение вершин, при котором вершины, от которых идут ребра к другим вершинам, предшествуют им в списке.

13. Может ли дерево содержать циклы?
14. По какому критерию программа `topo.java` (см. листинг 13.4) делает вывод о наличии в графе цикла?

Упражнения

Упражнения помогут вам глубже усвоить материал этой главы. Программирования они не требуют.

1. Создайте в приложении GraphN Workshop граф с пятью вершинами и семью ребрами. Запишите матрицу смежности графа, не используя кнопку `View`. Потом нажмите кнопку `View` и проверьте правильность своей матрицы.
2. Классическая игра «крестики-нолики» играется на доске 3 × 3; для простоты мы рассмотрим игру на доске 2 × 2, в которой игроку для победы достаточно выстроить в ряд два крестика или нолика. Используйте приложение GraphN для построения графа, моделирующего игру на доске 2 × 2. Необходима ли этому графу глубина 4?
3. Создайте матрицу смежности из пяти вершин, заполните ее случайными 0 и 1. Не обращайте внимания на симметрию. Теперь без использования кнопки `View` создайте соответствующий направленный граф в приложении GraphD Workshop. Затем нажмите кнопку `View` и проверьте, соответствует ли граф матрице смежности.
4. Попробуйте создать в приложении GraphD Workshop граф с циклом, который не будет распознан методом `topo()`.

Программные проекты

В этом разделе читателю предлагаются программные проекты, которые укрепляют понимание материала и демонстрируют практическое применение концепций этой главы.

13.1. Измените программу `bfs.java` (см. листинг 13.2) так, чтобы для поиска минимального остовного дерева применялся алгоритм обхода в ширину вместо обхода в глубину из программы `mst.java` (см. листинг 13.3). Создайте в методе `main()` граф с 9 вершинами и 12 ребрами, постройте его минимальное остовное дерево.

13.2. Измените программу `dfs.java` (см. листинг 13.1), чтобы вместо матрицы смежности в ней использовались списки смежности. Для работы со списками

можно воспользоваться классами `Link` и `LinkList` из программы `linkList2.java` (см. листинг 5.2) главы 5. Измените метод `find()` класса `LinkList`, чтобы он искал непосещенную вершину вместо значения ключа.

13.3. Измените программу `dfs.java` (см. листинг 13.1), чтобы она выводила таблицу связности для направленного графа (см. раздел «Связность в направленных графах»).

13.4. Реализуйте алгоритм Уоршелла для построения транзитивного замыкания графа. Начните с кода программного проекта 13.3. Полезно предусмотреть возможность вывода матрицы смежности на разных стадиях работы алгоритма.

13.5. «Ход коня» — древняя и хорошо известная шахматная головоломка. Шахматный конь делает ходы на пустой шахматной доске, пока каждое поле не будет посещено ровно один раз. Напишите программу, которая решает эту головоломку методом обхода в глубину. Размер доски желательно сделать переменным, чтобы решение можно было опробовать на доске меньшего размера. Для стандартной доски 8 × 8 расчет решения на настольном компьютере может занять несколько лет, но для доски 5 × 5 потребуется около минуты. За дополнительной информацией обращайтесь к разделу «Обход в глубину в программировании игр». Возможно, при совершении хода проще создавать нового коня, остающегося на новой клетке. При таком подходе конь будет соответствовать вершине, а последовательность создания коней может храниться в стеке. Игрок побеждает, когда вся доска будет заполнена конями (то есть при заполнении стека). Традиционно в этой задаче поля доски нумеруются последовательно, от 1 в левом верхнем углу до 64 в правом нижнем углу (или от 1 до 25 на доске 5 × 5). При поиске следующего хода конь должен не только сделать ход по правилам, но и не выйти за пределы доски или попасть на уже занятое (посещенное ранее) поле. Если после каждого хода программа будет выводить состояние доски и ожидать нажатия клавиши, вы сможете понаблюдать за работой алгоритма: как на доске появляется все больше коней, а в тупиковой ситуации программа снимает часть коней и опробует другую последовательность ходов. Сложность этой задачи более подробно анализируется в следующей главе.

Глава 14
Взвешенные графы

В предыдущей главе было показано, что ребра графа могут иметь направление. В этой главе рассматривается другая характеристика ребер: вес. Например, если вершины взвешенного графа соответствуют городам, то вес ребер может представлять расстояние между городами, стоимость перелета между ними или количество ежегодных автомобильных поездок (важно для проектировщиков дорожной сети).

При включении весов как характеристики ребер графа появляется ряд интересных и сложных вопросов. Что представляет собой минимальное остовное дерево взвешенного графа? Как вычислить наименьшее (или наименее затратное) расстояние от одной вершины к другой? Такие вопросы находят важное практическое применение в реальном мире.

Глава начинается с рассмотрения взвешенных, но не направленных графов и их минимальных остовных деревьев. Вторая половина главы посвящена графам, которые одновременно являются направленными и взвешенными, в контексте известного алгоритма Дейкстры, предназначенного для определения кратчайшего пути от одной вершины к другой.

Минимальное остовное дерево во взвешенных графах

Чтобы дать представление об особенностях взвешенных графов, мы вернемся к теме минимальных остовных деревьев. Для взвешенного графа построить такое дерево несколько сложнее, чем для невзвешенного. Когда все ребра имеют одинаковый вес, задача решается относительно тривиально (как было показано в главе 13, «Графы») — алгоритм легко выбирает ребро, включаемое в минимальное остовное дерево. Но если ребра имеют разные веса, правильный выбор потребует некоторых вычислений.

Пример: кабельное телевидение в джунглях

Предположим, мы хотим проложить линию кабельного телевидения, соединяющую шесть городов некоей вымышленной страны. Для соединения шести городов потребуется пять сегментов, но каких именно? Затраты на соединение разных

пар городов зависят от конкретной пары, поэтому мы должны тщательно выбрать маршрут для минимизации общих затрат.

На рис. 14.1 изображен взвешенный граф с шестью вершинами, представляющими города. Рядом с каждым ребром указан его вес. Будем считать, что эти числа представляют затраты на прокладку кабеля между городами. Обратите внимание: для некоторых пар прокладка кабеля невозможна из-за сложного рельефа или большого расстояния. Мы будем считать, что от A до C или от D до E слишком далеко, поэтому такие сегменты не нужно ни рассматривать, ни отображать их на графе.

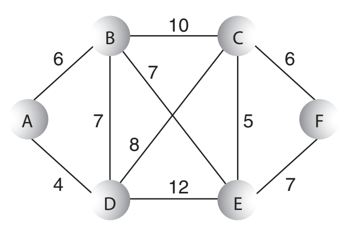


Рис. 14.1. Взвешенный граф

Как выбрать маршрут с минимальными общими затратами на прокладку кабеля? Для этого следует построить минимальное остовное дерево. Оно состоит из пяти сегментов (на единицу меньше количества городов), соединяет все шесть городов, а также обладает минимальными суммарными затратами. Сможете ли вы предложить такой маршрут по внешнему виду графа на рис. 14.1? Если не сможете, для решения задачи можно воспользоваться приложением GraphW Workshop.

Приложение GraphW Workshop

Приложение GraphW Workshop похоже на GraphN и GraphD, но оно создает взвешенные ненаправленные графы. Перед созданием ребра необходимо ввести его вес в текстовом поле, находящемся в правом верхнем углу. Приложение реализует только один алгоритм: при повторных нажатиях кнопки `Tree` оно определяет минимальное остовное дерево для созданного графа. Кнопки `New` и `View`, как и в предыдущих приложениях, предназначены для уничтожения старого графа и просмотра матрицы смежности.

Поэкспериментируйте с приложением, создайте небольшие графы и определите их минимальные остовные деревья. (В некоторых конфигурациях придется повозиться с расположением вершин, чтобы веса не накладывались друг на друга.)

В процессе работы алгоритма вершины, включаемые в минимальное остовное дерево, выделяются красным контуром, а ребра становятся толще. Вершины, находящиеся в дереве, также перечисляются слева под графом. Справа выводится содержимое приоритетной очереди. Например, элемент AB6 в очереди обозначает ребро из A в B с весом 6. Функции приоритетной очереди будут рассмотрены после примера алгоритма.

Создайте в приложении GraphW Workshop граф на рис. 14.1. Примерный вид результата показан на рис. 14.2.

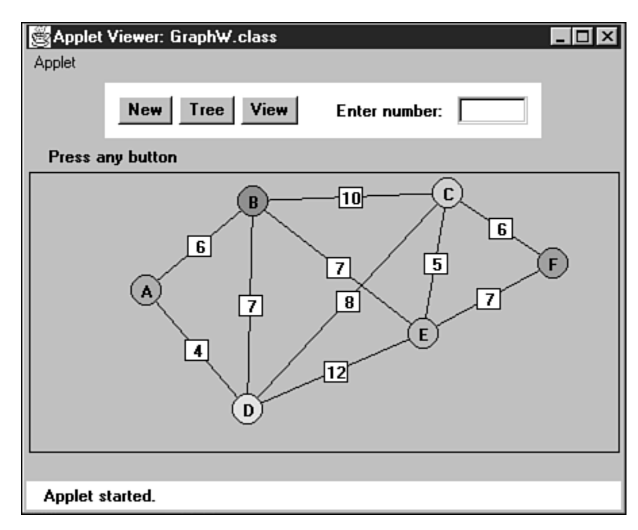


Рис. 14.2. Приложение GraphW Workshop

Постройте минимальное остовное дерево графа, выполняя алгоритм в пошаговом режиме (кнопка `Tree`). В результате должно получиться дерево, изображенное на рис. 14.3.

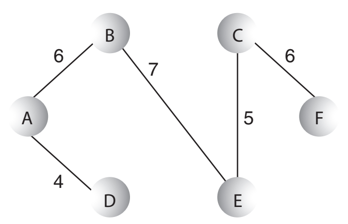


Рис. 14.3. Минимальное остовное дерево

Приложение должно обнаружить, что минимальное остовное дерево состоит из ребер AD, AB, BE, EC и CF с суммарным весом 28. Порядок перечисления ребер не важен. Если начать с другой вершины, получится дерево из тех же ребер, но в другом порядке.

Рассылка инспекторов

Алгоритм построения минимального остовного дерева непрост, поэтому для его рассмотрения мы воспользуемся аналогией с кабельным телевидением. В фирме кроме начальника (это, конечно, вы) работают еще несколько топографов-инспекторов.

Компьютерный алгоритм не знает всех данных задачи и не умеет представить себе общую картину. Он накапливает данные постепенно, изменяя свое представление о ситуации по ходу дела. При работе с графами алгоритмы обычно начи-

нают с определенной вершины и продвигаются «наружу», получая информацию о близлежащих вершинах раньше информации о дальних вершинах. Примеры такого подхода уже встречались нам в описаниях обхода в глубину и в ширину (см. предыдущую главу).

Аналогичным образом мы будем считать, что затраты на прокладку кабеля между всеми городами неизвестны заранее. Получение этой информации требует времени, и в этом нам помогут инспекторы.

Начинаем с вершины A

Для начала мы открываем офис в городе A (впрочем, начать можно с любого города). Из A доступны только два города: B и D (см. рис. 14.1). Вы отправляете двух инспекторов по опасным лесным тропам: первый идет в B, а второй в D. Задача инспекторов — определить стоимость прокладки кабеля по этим маршрутам.

Первый инспектор прибывает в B, завершив осмотр местности, и звонит вам; он говорит, что прокладка кабеля между A и B обойдется в 6 миллионов долларов. Второй инспектор чуть позднее сообщает, что сегмент A-D, проходящий по равнинной части страны, будет стоить только 4 миллиона. Вы составляете список:

- A-D, 4 миллиона.
- A-B, 6 миллионов.

Сегменты всегда перечисляются в порядке возрастания стоимости; причины вскоре будут понятны.

Прокладка сегмента A-D

На этой стадии можно отправлять строителей на прокладку кабеля из A в D. Почему мы уверены в том, что маршрут A-D станет частью самого экономичного решения (минимального остовного дерева)? Пока нам известна стоимость всего двух сегментов системы. Разве нам не нужна дополнительная информация?

Попробуйте представить, что какой-то другой маршрут, связывающий A с D, окажется дешевле прямого соединения. Если он не идет непосредственно в D, то этот другой маршрут должен пройти через B, а может быть, и через другие города. Но мы уже знаем, что сегмент в B (6) стоит дороже сегмента в D (4). Таким образом, даже если остальные сегменты этого гипотетического кругового маршрута будут дешевыми (рис. 14.4), переход в D через B все равно обойдется дороже. Кроме того, переход к городам на обходном пути (X) через B обойдется дороже, чем переход через D.

Итак, мы делаем вывод, что сегмент A-D должен быть частью минимального остовного дерева. Заодно это наводит на мысль, что в качестве отправной точки лучше всего выбрать самый дешевый сегмент (формальное доказательство выходит за рамки книги). Мы строим сегмент A-D и открываем офис в D.

Зачем нужен офис? По местному законодательству отправка инспекторов в близлежащие города возможна только при наличии офиса. А в контексте теории графов это означает, что вершина должна быть включена в дерево перед определением весов ребер, выходящих из этой вершины. Все города с офисами соединены друг с другом проложенными кабелями; города без офисов еще не подключены к сети.

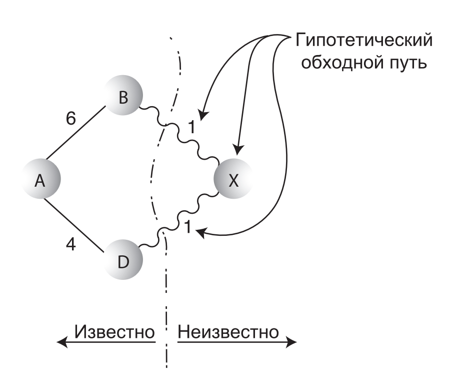


Рис. 14.4. Гипотетический обходной путь

Прокладка сегмента A-B

После завершения сегмента A-D и открытия офиса в D можно отправить инспекторов по всем городам, доступным из D. Это города B, C и E. Инспекторы добираются до места назначения и сообщают стоимость работ: 7, 8 и 12 миллионов соответственно. (Конечно, отправлять инспектора в A не нужно, потому что мы уже проанализировали маршрут A-D и проложили кабель.)

Теперь мы знаем стоимость прокладки четырех сегментов из городов с офисами в города без офисов:

- A-B, 6 миллионов.
- D-B, 7 миллионов.
- D-C, 8 миллионов.
- D-E, 12 миллионов.

Почему в списке нет сегмента A-D? Потому что кабель между этими городами уже проложен; этот участок сети можно больше не рассматривать. Маршрут, по которому был проложен канал, всегда удаляется из списка.

Что же делать дальше? Имеется множество потенциальных сегментов. Ваши дальнейшие действия должны определяться следующим правилом.

ПРАВИЛО

Из списка всегда выбирается ребро с наименьшим весом.

Вообще говоря, мы уже следовали этому правилу при выборе маршрута из A; ребро A-D было самым «дешевым». В текущей ситуации наименьшим весом обладает ребро A-B, поэтому мы прокладываем кабель из A в B за 6 миллионов и открываем офис в B.

Здесь необходимо сделать одно общее замечание. В любой момент при проектировании кабельной сети существуют три категории городов:

1. Города, в которых имеются офисы, связанные кабелем. (В терминологии графов — вершины, входящие в минимальное остовное дерево.)

2. Города, еще не подключенные к сети и не имеющие офисов, но для которых известна стоимость соединения как минимум с одним городом, в котором существует офис. Мы будем называть их «периферийными» городами.
3. Города, о которых ничего не известно.

На данный момент A, D и B относятся к категории 1, C и E — к категории 2, а F — к категории 3 (рис. 14.5). В ходе работы алгоритма города переходят из категории 3 в 2, а из категории 2 в категорию 1.

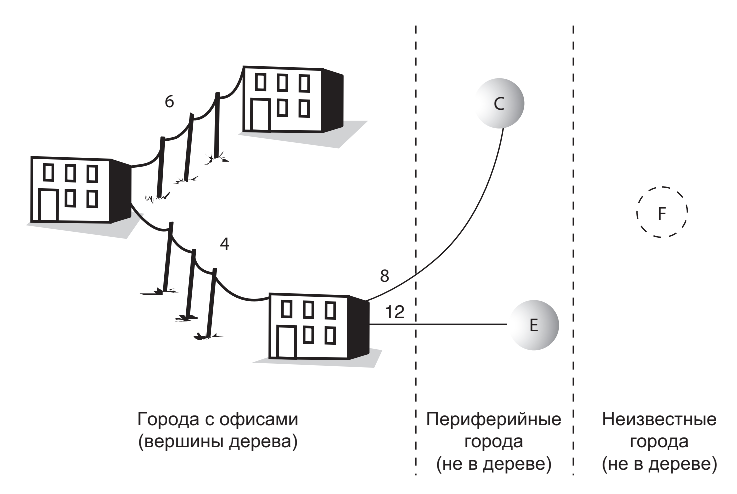


Рис. 14.5. Промежуточная стадия построения минимального остовного дерева

Прокладка сегмента B-E

К этому моменту города A, D и B подключены к кабельной сети и в них открыты офисы. Мы уже знаем стоимость прокладки кабеля из A и D в города категории 2, но для B затраты пока неизвестны. Инспекторы из B отправляются в C и E. От них приходит информация: прокладка кабеля в C обойдется в 10 миллионов, а в E — 7 миллионов. Новый список выглядит так:

- ✦ B-E, 7 миллионов.
- ✦ D-C, 8 миллионов.
- ✦ B-C, 10 миллионов.
- ✦ D-E, 12 миллионов.

Сегмент D-B был в предыдущем списке, но в новом варианте списка его нет. Как упоминалось ранее, бессмысленно прокладывать сегменты к уже подключенным городам.

Из списка видно, что наименьшим весом обладает маршрут B-E (7). Вы отправляете рабочих на прокладку кабеля и открываете офис в E (см. рис. 14.3).

Прокладка сегмента E-C

Из E инспекторы сообщают, что стоимость прокладки кабеля в C составляет 5 миллионов, а в F — 7 миллионов. Сегмент D-E удаляется из списка, так как город E уже подключен. Обновленный список выглядит так:

- ✦ E-C, 5 миллионов.
- ✦ E-F, 7 миллионов.
- ✦ D-C, 8 миллионов.
- ✦ B-C, 10 миллионов.

Наименьшей стоимостью обладает сегмент E-C. Вы строите его и открываете офис в C.

Последний сегмент C-F

Количество вариантов постоянно сокращается. После удаления подключенных городов в списке остаются только два сегмента:

- ✦ C-F, 6 миллионов.
- ✦ E-F, 7 миллионов.

Вы прокладываете последний сегмент кабеля из C в F, открываете офис в F — работа завершена. В каждом городе имеется офис, а между городами проложена кабельная сеть A-D, A-B, B-E, E-C и C-F (см. рис. 14.3). У нас получилась сеть, соединяющая все шесть городов с минимальными затратами.

Создание алгоритма

На примере прокладки сети кабельного телевидения были представлены основные принципы построения минимальных остовных деревьев для взвешенных графов. Теперь посмотрим, как алгоритмизируется этот процесс.

Приоритетная очередь

Как видно из примера с кабельной сетью, основной операцией в этом алгоритме является ведение списка стоимости сегментов между парами городов. Выбор сегмента с минимальной стоимостью определял место прокладки следующего участка.

Для списка, из которого требуется раз за разом выбирать наименьшее значение, логично использовать приоритетную очередь. Оказывается, эта структура данных весьма эффективна при построении минимального остовного дерева. В серьезной программе приоритетная очередь может быть реализована на базе пирамиды (см. главу 12, «Пирамиды»); это ускорит выполнение операций с большими очередями. В нашей демонстрационной программе будет использоваться приоритетная очередь на базе простого массива.

Общая схема алгоритма

Сформулируем алгоритм в терминологии графов (вместо терминологии кабельной сети).

Начать с вершины, включить ее в дерево. Затем многократно выполнять следующие действия:

1. Найти все ребра от самой новой вершины к другим вершинам, не входящим в дерево. Поместить эти ребра в приоритетную очередь.
2. Выбрать ребро с наименьшим весом. Включить его и вершину, к которой оно ведет, в дерево.

Действия повторяются до тех пор, пока все вершины не будут включены в дерево. При выполнении этого условия работа алгоритма завершается.

На шаге 1 под «самой новой» понимается вершина, которая была последней добавлена в дерево. Поиск ребер для этого шага осуществляется в матрице смежности. После шага 1 в списке содержатся все ребра, ведущие от вершин в дереве к периферийным вершинам.

Избыточные ребра

При составлении списка мы позаботились об удалении сегментов, ведущих к уже соединенным городам. Без этого в сети могли бы появиться лишние сегменты.

В реализации алгоритма также следует проследить за исключением из приоритетной очереди ребер, ведущих к вершинам, которые уже включены в дерево. Можно просматривать содержимое очереди и удалять все такие ребра каждый раз, когда в дерево добавляется новая вершина. Но как выясняется, проще в любой момент времени хранить в приоритетной очереди только одно ребро к заданной периферийной вершине. Другими словами, очередь должна содержать только одно ребро, ведущее к каждой вершине категории 2.

Именно это решение используется в приложении GraphW Workshop. Приоритетная очередь содержит меньше ребер, чем можно ожидать — всего один элемент для каждой вершины категории 2. Пройдите по минимальному остовному дереву для рис. 14.1 и убедитесь в том, что это действительно так. В таблице 14.1 показано, как ребра с повторяющимися «конечными точками» удаляются из приоритетной очереди.

Таблица 14.1. Отсечение ребер

Номер шага	Полный список ребер	Усеченный список (в приоритетной очереди)	Дубликаты, удаленные из приоритетной очереди
1	AB6, AD4	AB6, AD4	
2	DE12, DC8, DB7, AB6	DE12, DC8, AB6	DB7(AB6)
3	DE12, BC10, DC8, BE7	DC8, BE7	DE12(BE7), BC10(DC8)
4	BC10, DC8, EF7, EC5	EF7, EC5	BC10(EC5), DC8(EC5)
5	EF7, CF6	CF6	EF7

Напомним, что ребро определяется начальной вершиной, конечной вершиной и весом. Второй столбец таблицы соответствует тем спискам, которые мы вели при построении кабельной сети. В нем перечисляются все ребра от вершин категории 1 (входящих в дерево) к вершинам категории 2 (для которых известно хотя бы одно ребро, ведущее от вершины категории 1).

В третьем столбце приводится содержимое приоритетной очереди при выполнении приложения GraphW. Из очереди удаляются все ребра с такой же конечной вершиной, как у другого ребра, но с бóльшим весом.

В четвертом столбце перечислены удаленные ребра; в круглых скобках указаны ребра с меньшим весом, которые остались в очереди вместо них. При переходе к следующему шагу из списка всегда исключается последний элемент, потому что соответствующее ребро включается в очередь.

Поиск дубликатов в приоритетной очереди

Как сделать так, чтобы для каждой вершины категории 2 в очереди было только одно ребро? Каждый раз, когда ребро включается в очередь, мы убеждаемся в отсутствии другого ребра, ведущего к той же вершине. Если такое ребро существует, то из двух ребер остается только одно, имеющее минимальный вес.

Из этого следует, что алгоритм должен проверять приоритетную очередь элемент за элементом в поисках ребер-дубликатов. Произвольный доступ в приоритетных очередях выполняется неэффективно, однако в данной ситуации эта операция, противоречащая «духу» приоритетной очереди, необходима.

Реализация на языке Java

Метод построения минимального остовного дерева для взвешенного графа mstw() работает по описанному выше алгоритму. Как и в других программах, работающих с графами, список вершин хранится в массиве vertexList[], а первая вершина хранится в ячейке с индексом 0. Переменная currentVert представляет вершину, которая была последней включена в дерево.

```
public void mstw()           // Построение минимального остовного дерева
   {
   currentVert = 0;          // Начиная с ячейки 0

   while(nTree < nVerts-1)   // Пока не все вершины включены в дерево
      {                      // Включение currentVert в дерево
      vertexList[currentVert].isInTree = true;
      nTree++;

      // Вставка в приоритетную очередь ребер, смежных с currentVert
      for(int j=0; j<nVerts; j++)   // Для каждой вершины
         {
         if(j==currentVert)         // Пропустить, если текущая вершина
            continue;
         if(vertexList[j].isInTree) // Пропустить, если уже в дереве
```

```
         continue;
      int distance = adjMat[currentVert][j];
      if( distance == INFINITY)  // Пропустить, если нет ребер
         continue;
      putInPQ(j, distance);      // Поместить в приоритетную очередь
      }
   if(thePQ.size()==0)           // Очередь не содержит вершин?
      {
      System.out.println(" GRAPH NOT CONNECTED");
      return;
      }
      // Удаление ребра с минимальным расстоянием из очереди
      Edge theEdge = thePQ.removeMin();
      int sourceVert = theEdge.srcVert;
      currentVert = theEdge.destVert;

      // Вывод ребра от начальной до текущей вершины
      System.out.print( vertexList[sourceVert].label );
      System.out.print( vertexList[currentVert].label );
      System.out.print(" ");
      }

   // Минимальное остовное дерево построено
   for(int j=0; j<nVerts; j++)     // Снятие пометки с вершин
      vertexList[j].isInTree = false;
   }
```

Алгоритм выполняется в цикле `while`, который завершается с включением всех вершин в дерево. В цикле выполняются следующие действия:

1. Текущая вершина включается в дерево.
2. Ребра, смежные с этой вершиной, включаются в приоритетную очередь (если требуется).
3. Ребро с минимальным весом исключается из приоритетной очереди. Конечная вершина этого ребра становится текущей вершиной.

Рассмотрим все эти действия более подробно. На шаге 1 в дерево включается вершина `currentVert` (посредством установки флага `isInTree`).

На шаге 2 вершины, смежные с этой вершиной, анализируются в качестве кандидатов на вставку в приоритетную очередь. Ребра перебираются по элементам строки матрицы смежности, номер которой равен `currentVert`. Ребро включается в очередь, если не выполняется ни одно из следующих условий:

- Начальная вершина совпадает с конечной.
- Конечная вершина включена в дерево.
- Не существует ребра, ведущего к этой вершине.

Если ни одно из этих условий не выполняется, то вызывается метод `putInPQ()` для включения ребра в приоритетную очередь. Впрочем, как мы вскоре убедимся, этот метод тоже не всегда включает ребро в очередь.

На шаге 3 ребро с минимальным весом исключается из приоритетной очереди. Это ребро и его конечная вершина включаются в дерево, а начальная (currentVert) и приемная вершины выводятся приложением.

В конце выполнения mstw() вершины исключаются из дерева, для чего метод сбрасывает их переменные isInTree. Этот шаг не является строго обязательным, потому что на основе данных строится только одно дерево. И все же хороший стиль программирования рекомендует возвращать данные к исходному состоянию после их использования.

Как упоминалось ранее, приоритетная очередь должна содержать только одно ребро с заданной конечной вершиной. Метод putInPQ() обеспечивает выполнение этого условия. Он вызывает метод find() класса PriorityQ, адаптированный для поиска ребра с заданной конечной вершиной. Если такого ребра нет, а метод find() соответственно возвращает −1, то метод putInPQ() просто вставляет ребро в приоритетную очередь. Но если такое ребро существует, putInPQ() проверяет, какое из двух ребер — существующее или новое — обладает меньшим весом. Если вес меньше у старого ребра, изменения не требуются, а если у нового — старое ребро выводится из очереди, а новое ребро включается в нее. Код метода putInPQ():

```
public void putInPQ(int newVert, int newDist)
   {
   // Существует ли другое ребро с той же конечной вершиной?
   int queueIndex = thePQ.find(newVert);  // Получение индекса
   if(queueIndex != -1)              // Если ребро существует,
      {                              // получить его
      Edge tempEdge = thePQ.peekN(queueIndex);
      int oldDist = tempEdge.distance;
      if(oldDist > newDist)          // Если новое ребро короче,
         {
         thePQ.removeN(queueIndex);  // удалить старое ребро
         Edge theEdge = new Edge(currentVert, newVert, newDist);
         thePQ.insert(theEdge);      // Вставка нового ребра
         }
      // Иначе ничего не делается; оставляем старую вершину
      }
   else  // Ребра с той же конечной вершиной не существует
      {                              // Вставка нового ребра
      Edge theEdge = new Edge(currentVert, newVert, newDist);
      thePQ.insert(theEdge);
      }
   }
```

Программа mstw.java

Класс PriorityQ использует массив для хранения своих элементов. Как упоминалось ранее, в программе, работающей с большими графами, вместо массива лучше использовать пирамиду. Класс PriorityQ был дополнен вспомогательными методами. Уже упоминавшийся метод find() ищет ребро с заданной конечной вершиной. Метод peekN() читает значение произвольного элемента, а метод removeN() удаляет

произвольный элемент. В остальном код класса выглядит уже знакомо. В листинге 14.1 приведен полный код программы `mstw.java`.

Листинг 14.1. Программа mstw.java

```
// mstw.java
// Построение минимального остовного дерева для взвешенного графа
// Запуск программы: C>java MSTWApp
////////////////////////////////////////////////////////////////
class Edge
   {
   public int srcVert;   // Индекс начальной вершины ребра
   public int destVert;  // Индекс конечной вершины ребра
   public int distance;  // Расстояние от начала до конца
// -------------------------------------------------------------
   public Edge(int sv, int dv, int d)  // Конструктор
      {
      srcVert = sv;
      destVert = dv;
      distance = d;
      }
// -------------------------------------------------------------
   }  // Конец класса Edge
////////////////////////////////////////////////////////////////
   class PriorityQ
      {
      // Массив отсортирован по убыванию от ячейки 0 до size-1
      private final int SIZE = 20;
      private Edge[] queArray;
      private int size;
// -------------------------------------------------------------
   public PriorityQ()            // Конструктор
      {
      queArray = new Edge[SIZE];
      size = 0;
      }
// -------------------------------------------------------------
   public void insert(Edge item)  // Вставка элемента в порядке сортировки
      {
      int j;

      for(j=0; j<size; j++)           // Поиск места для вставки
         if( item.distance >= queArray[j].distance )
            break;

      for(int k=size-1; k>=j; k--)    // Перемещение элементов вверх
         queArray[k+1] = queArray[k];

      queArray[j] = item;             // Вставка элемента
      size++;
      }
```

продолжение ⇨

Листинг 14.1 (*продолжение*)

```
// -------------------------------------------------------------
   public Edge removeMin()            // Извлечение наименьшего элемента
      { return queArray[--size]; }
// -------------------------------------------------------------
   public void removeN(int n)         // Удаление элемента в позиции N
      {
      for(int j=n; j<size-1; j++)     // Перемещение элементов вниз
         queArray[j] = queArray[j+1];
      size--;
      }
// -------------------------------------------------------------
   public Edge peekMin()          // Чтение наименьшего элемента
      { return queArray[size-1]; }
// -------------------------------------------------------------
   public int size()              // Получение количества элементов
      { return size; }
// -------------------------------------------------------------
   public boolean isEmpty()      // true, если очередь пуста
      { return (size==0); }
// -------------------------------------------------------------
   public Edge peekN(int n)      // Чтение элемента в позиции N
      { return queArray[n]; }
// -------------------------------------------------------------
   public int find(int findDex)  // Поиск элемента с заданным
      {                          // значением destVert
      for(int j=0; j<size; j++)
         if(queArray[j].destVert == findDex)
            return j;
      return -1;
      }
// -------------------------------------------------------------
   }  // Конец класса PriorityQ
////////////////////////////////////////////////////////////////
class Vertex
   {
   public char label;        // Метка (например, 'A')
   public boolean isInTree;
// -------------------------------------------------------------
   public Vertex(char lab)   // Конструктор
      {
      label = lab;
      isInTree = false;
      }
// -------------------------------------------------------------
   }  // Конец класса Vertex
////////////////////////////////////////////////////////////////
class Graph
   {
   private final int MAX_VERTS = 20;
```

```
   private final int INFINITY = 1000000;
   private Vertex vertexList[]; // Список вершин
   private int adjMat[][];      // Матрица смежности
   private int nVerts;          // Текущее количество вершин
   private int currentVert;
   private PriorityQ thePQ;
   private int nTree;           // Количество вершин в дереве
// -------------------------------------------------------------
   public Graph()               // Конструктор
      {
      vertexList = new Vertex[MAX_VERTS];
                                // Матрица смежности
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      for(int j=0; j<MAX_VERTS; j++)      // Матрица смежности
         for(int k=0; k<MAX_VERTS; k++)   // заполняется нулями
            adjMat[j][k] = INFINITY;
      thePQ = new PriorityQ();
   }
// -------------------------------------------------------------
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
      }
// -------------------------------------------------------------
   public void addEdge(int start, int end, int weight)
      {
      adjMat[start][end] = weight;
      adjMat[end][start] = weight;
      }
// -------------------------------------------------------------
   public void displayVertex(int v)
      {
      System.out.print(vertexList[v].label);
      }
// -------------------------------------------------------------
   public void mstw()        // Построение минимального остовного дерева
      {
      currentVert = 0;          // Начиная с ячейки 0

      while(nTree < nVerts-1)   // Пока не все вершины включены в дерево
         {                      // Включение currentVert в дерево
         vertexList[currentVert].isInTree = true;
         nTree++;

         // Вставка в приоритетную очередь ребер, смежных с currentVert
         for(int j=0; j<nVerts; j++)   // Для каждой вершины
            {
            if(j==currentVert)         // Пропустить, если текущая вершина
```

продолжение ⇗

Листинг 14.1 (*продолжение*)

```
            continue;
         if(vertexList[j].isInTree) // Пропустить, если уже в дереве
            continue;
         int distance = adjMat[currentVert][j];
         if( distance == INFINITY)  // Пропустить, если нет ребер
            continue;
         putInPQ(j, distance);      // Поместить в приоритетную очередь
         }
      if(thePQ.size()==0)           // Очередь не содержит вершин?
         {
         System.out.println(" GRAPH NOT CONNECTED");
         return;
         }
         // Удаление ребра с минимальным расстоянием из очереди
         Edge theEdge = thePQ.removeMin();
         int sourceVert = theEdge.srcVert;
         currentVert = theEdge.destVert;

         // Вывод ребра от начальной до текущей вершины
         System.out.print( vertexList[sourceVert].label );
         System.out.print( vertexList[currentVert].label );
         System.out.print(" ");
         }

      // Минимальное остовное дерево построено
      for(int j=0; j<nVerts; j++)     // Снятие пометки с вершин
         vertexList[j].isInTree = false;
      }
// -------------------------------------------------------------
   public void putInPQ(int newVert, int newDist)
      {
      // Существует ли другое ребро с той же конечной вершиной?
      int queueIndex = thePQ.find(newVert);  // Получение индекса
      if(queueIndex != -1)              // Если ребро существует,
         {                              // получить его
         Edge tempEdge = thePQ.peekN(queueIndex);
         int oldDist = tempEdge.distance;
         if(oldDist > newDist)          // Если новое ребро короче,
            {
            thePQ.removeN(queueIndex);  // удалить старое ребро
            Edge theEdge = new Edge(currentVert, newVert, newDist);
            thePQ.insert(theEdge);      // Вставка нового ребра
            }
         // Иначе ничего не делается; оставляем старую вершину
         }
      else  // Ребра с той же конечной вершиной не существует
         {                              // Вставка нового ребра
         Edge theEdge = new Edge(currentVert, newVert, newDist);
```

```
         thePQ.insert(theEdge);
         }
      }
// -------------------------------------------------------------
   }  // Конец класса Graph
////////////////////////////////////////////////////////////////
class MSTWApp
   {
   public static void main(String[] args)
      {
      Graph theGraph = new Graph();
      theGraph.addVertex('A');    // 0  (исходная вершина)
      theGraph.addVertex('B');    // 1
      theGraph.addVertex('C');    // 2
      theGraph.addVertex('D');    // 3
      theGraph.addVertex('E');    // 4
      theGraph.addVertex('F');    // 5

      theGraph.addEdge(0, 1, 6);  // AB  6
      theGraph.addEdge(0, 3, 4);  // AD  4
      theGraph.addEdge(1, 2, 10); // BC 10
      theGraph.addEdge(1, 3, 7);  // BD  7
      theGraph.addEdge(1, 4, 7);  // BE  7
      theGraph.addEdge(2, 3, 8);  // CD  8
      theGraph.addEdge(2, 4, 5);  // CE  5
      theGraph.addEdge(2, 5, 6);  // CF  6
      theGraph.addEdge(3, 4, 12); // DE 12
      theGraph.addEdge(4, 5, 7);  // EF  7

      System.out.print("Minimum spanning tree: ");
      theGraph.mstw();            // Минимальное остовное дерево
      System.out.println();
      }
   }  // Конец класса MSTWApp
///////////////////////////////////////////////////////////////
```

Метод `main()` класса `MSTWApp` создает дерево, изображенное на рис. 14.1. Выходные данные выглядят так:

```
Minimum spanning tree: AD AB BE EC CF
```

Задача выбора кратчайшего пути

Пожалуй, самой распространенной задачей, связанной с взвешенными графами, является задача выбора кратчайшего пути между двумя вершинами. Решение этой задачи находит практическое применение во множестве реальных ситуаций, от проектирования печатных плат до планирования проектов. Эта задача превосходит по сложности то, что мы делали раньше, поэтому мы начнем с рассмотрения примера.

Железная дорога

Действие происходит все в той же вымышленной стране, но на этот раз нас интересует не кабельное телевидение, а железная дорога. Впрочем, новый проект не столь грандиозен, как предыдущий — железную дорогу строить не нужно; она уже построена. Мы всего лишь хотим найти самый экономичный маршрут из одного города в другой.

Стоимость билета между любыми двумя городами является фиксированной величиной. Данные представлены на рис. 14.6. Таким образом, поездка из A в B будет стоить $50, поездка из B в D обойдется в $90 и т. д. Цена не зависит от того, является ли поездка сегментом более длинного маршрута или нет (в отличие от современных авиаперелетов).

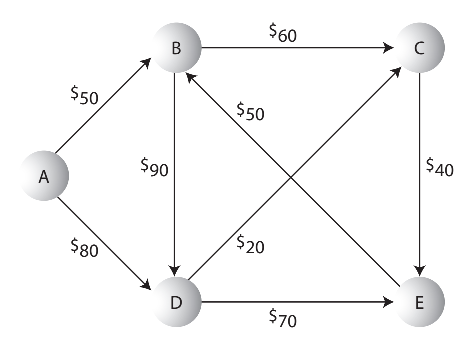


Рис. 14.6. Стоимость поездок

Граф на рис. 14.6 имеет направленные ребра. Они представляют одноколейные железнодорожные линии, по которым (в интересах безопасности) движение осуществляется только в одну сторону. Например, из A в B можно проехать напрямую, а из B в A прямой дороги нет. Хотя в данной ситуации нужно найти самый экономичный маршрут, задача называется задачей выбора кратчайшего пути. Под «кратчайшим» подразумевается не только самый короткий, но и самый экономичный, самый быстрый или оптимальный по какой-то другой характеристике маршрут.

Между любыми двумя городами существует несколько маршрутов. Например, из A в E можно проехать через D, через B и C, через D и C или каким-нибудь другим путем. (Добраться до города F по железной дороге невозможно, поэтому на графе он не представлен. И это хорошо, потому что сокращение количества вершин сокращает размер некоторых списков, которые нам предстоит построить.)

Задача выбора кратчайшего пути в нашем случае формулируется так: какой маршрут будет самым экономичным для заданной начальной и конечной точек? При взгляде на рис. 14.6 нетрудно представить, что самый экономичный маршрут из A в E проходит через D и C и стоит $140.

Направленный взвешенный граф

Как упоминалось ранее, в стране проложена одноколейная железная дорога, поэтому перемещение между любыми двумя городами возможно только в одном направлении. Такая железнодорожная сеть моделируется направленным графом. В более реалистичной ситуации между любыми двумя городами можно было бы перемещаться в обоих направлениях за одинаковую цену; такая модель соответствовала бы ненаправленному графу. Впрочем, задача выбора кратчайшего пути в этих случаях решается сходным образом, поэтому для разнообразия мы покажем, как выглядит решение для направленного графа.

Алгоритм Дейкстры

Следующее решение задачи кратчайшего пути называется *алгоритмом Дейкстры* (по имени Э. Дейкстры, описавшего его в 1959 г.) Алгоритм основан на представлении графа в виде матрицы смежности. Интересно, что алгоритм Дейкстры находит не только кратчайший путь от одной заданной вершины к другой, но и кратчайшие пути от заданной вершины ко всем остальным вершинам.

Агенты и поездки

Чтобы понять, как работает алгоритм Дейкстры, представьте, что вы ищете самый дешевый вариант поездки из A в другие города. Вы (и нанятые вами агенты) будете выполнять функции компьютерной программы, реализующей алгоритм Дейкстры. Конечно, в реальной жизни вы попросту купили бы железнодорожный справочник со всеми ценами. Однако алгоритм должен получать информацию последовательно, поэтому (по аналогии с предыдущим разделом) мы будем считать, что полная информация изначально недоступна.

В каждом городе кассир может сказать, сколько будет стоить прямая поездка в другие города (то есть поездка, при которой не приходится делать пересадку). К сожалению, кассир не знает стоимости билетов в города, находящиеся на расстоянии более одного сегмента. Вы рисуете в блокноте таблицу со столбцом для каждого города. В конце работы алгоритма в каждом столбце должен быть построен самый дешевый маршрут от начальной точки до этого города.

Первый агент: A

В каждом городе должен появиться ваш агент, задача которого — получить информацию о стоимости билетов в другие города. В городе A таким агентом являетесь вы сами. Кассир в A знает лишь то, что поездка в B стоит $50, а поездка в D стоит $80. Вы записываете эту информацию в блокнот (табл. 14.2).

Таблица 14.2. Шаг 1: агент в A

Из города A	B	C	D	E
Шаг 1	50 (через A)	Бск	80 (через A)	Бск

Сокращение «Бск» означает «бесконечность»; из A нельзя попасть в город, указанный в заголовке столбца — или по крайней мере вы еще не знаете, как туда попасть. (Как вы вскоре увидите, в алгоритме бесконечность представляется очень большим числом для упрощения расчетов.) В круглых скобках указывается последний город, посещенный перед прибытием в разные конечные точки. Вскоре вы увидите, для чего нужна эта информация. Что делать теперь? Действуйте по следующему правилу.

ПРАВИЛО

Всегда отправляйте агента в город с минимальной общей стоимостью билета от начальной точки (A).

Города, в которых уже имеется агент, не рассматриваются. Обратите внимание на отличие этого правила от того, которое использовалось при построении минимального остовного дерева (прокладка кабельной сети). Там мы выбирали самый дешевый сегмент (ребро) от подключенных городов к неподключенным, а сейчас выбирается самый экономичный общий маршрут от A к городу без агента. В этой конкретной фазе нашего исследования эти два принципа приводят к одинаковым результатам, потому что все известные маршруты от A состоят лишь из одного ребра, но с рассылкой агентов по другим городам маршруты от A превратятся в набор сегментов.

Второй агент: B

Самый дешевый маршрут из A ведет в B ($50). Вы нанимаете работника и отправляете его в B, где он будет вашим агентом. Он звонит вам и сообщает, что по словам кассира B поездка в C будет стоить $60, а поездка в D обойдется в $90.

Несложные вычисления показывают, что перемещение из A в C через B будет стоить $50 + $60 = $110, поэтому вы изменяете запись для C. Также видно, что перемещение из A в D через B стоит $50 + $90 = $140. Однако — и это очень важный момент — мы уже знаем, что прямая поездка из A в D стоит всего $80. Нас интересует только самый экономичный маршрут из A, поэтому более дорогой маршрут игнорируется, а запись в таблице остается без изменений. В таблице 14.3 приведено текущее состояние таблицы, а на рис. 14.7 изображено его географическое представление.

Таблица 14.3. Шаг 2: агенты в A и B

Из города A	B	C	D	E
Шаг 1	50 (через A)	Бск	80 (через A)	Бск
Шаг 2	50 (через A)*	110 (через B)	80 (через A)	Бск

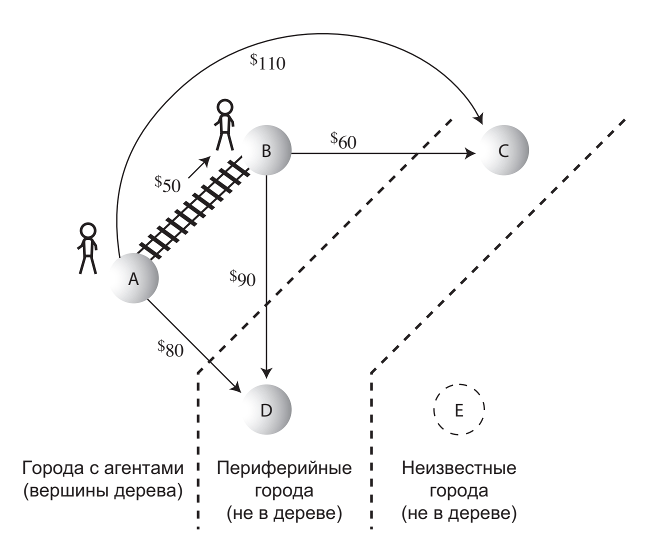


Рис. 14.7. Шаг 2 в алгоритме выбора кратчайшего пути

После того как в городе появится агент, мы точно знаем, что маршрут, выбранный этим агентом для перемещения в город, является самым экономичным. Почему? Возьмем текущий пример. Если бы существовал более дешевый маршрут для перемещения из A в B, то он должен был бы проходить через другой город. Но из A можно перейти только в D, а уже одна эта поездка обойдется дороже прямого перемещения в B. Добавление дополнительных маршрутов для перемещения из D в B только повысит стоимость маршрута в D. Из этого следует, что в дальнейшем самый экономичный маршрут из A в B не должен обновляться. Его стоимость останется неизменной, какую бы информацию мы ни получили о других городах. Маршрут помечается звездочкой (*) — это означает, что в городе имеется агент, а самый дешевый маршрут в него зафиксирован.

Три категории городов

В алгоритме минимального остовного дерева все города делятся на три категории:

1. Города с агентом (входящие в дерево).
2. Города с известной стоимостью перемещения из городов с агентами (периферийные города).
3. Неизвестные города.

В текущем состоянии города A и B относятся к категории 1, потому что там уже присутствуют агенты. Города категории 1 образуют дерево из путей, которые идут от начальной вершины к разным конечным вершинам. (Конечно, это дерево не совпадает с минимальным остовным деревом.)

В некоторых других городах агентов нет, но стоимость перемещения в них известна, потому что агенты находятся в смежных городах категории 1. Мы знаем, что поездка из A в D стоит $80, а поездка из B в C обойдется в $60. Так как стоимость

поездки в D и C известна, эти города относятся к категории 2 (периферийные города).

Пока мы не располагаем информацией о городе E, поэтому этот город относится к категории «неизвестных». На рис. 14.7 показано распределение городов по категориям для текущего состояния алгоритма.

Как и при построении минимального остовного дерева, этот алгоритм в процессе своей работы постепенно перемещает города из категории неизвестных в категорию периферийных, а из категории периферийных — в дерево.

Третий агент: D

На данный момент самым экономичным из известных маршрутов из A в любой город без агента является прямой маршрут A-D за $80. Маршруты A-B-C ($110) и A-B-D ($140) стоят дороже.

Вы нанимаете очередного агента и отправляете его в D за $80. Агент сообщает, что из D можно добраться в C ($20) или в E ($80). Теперь можно изменить данные для C. Ранее добраться до этого города из A можно было через B за $110. Теперь мы также видим, что до C можно добраться всего за $100 через D. Также определилась стоимость поездки из A в ранее неизвестный город E — $150 через D. Вы отмечаете внесенные изменения (табл. 14.4 и рис. 14.8).

Таблица 14.4. Шаг 3: агенты в A, B и D

Из города A	B	C	D	E
Шаг 1	50 (через A)	Бск	80 (через A)	Бск
Шаг 2	50 (через A)*	110 (через B)	80 (через A)	Бск
Шаг 3	50 (через A)*	100 (через D)	80 (через A)*	150 (через D)

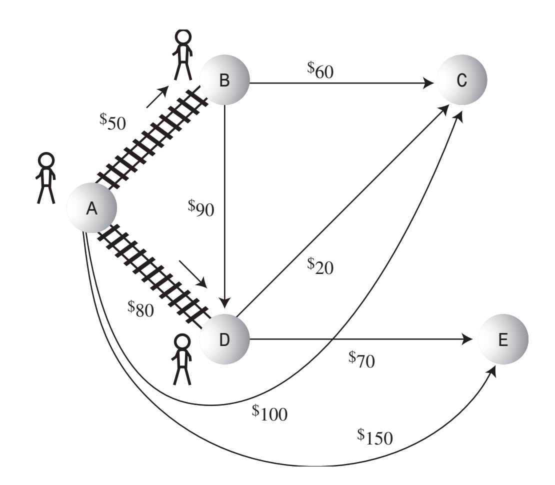


Рис. 14.8. Шаг 3 в алгоритме выбора кратчайшего пути

Четвертый агент: C

Теперь самым экономичным путем в любой город без агента становится путь из A в C через D за $100. Соответственно вы отправляете агента по этому маршруту в C. Агент сообщает, что из C в E можно добраться за $40. А раз C находится на расстоянии $100 из A (через D), а E находится на расстоянии $40 от C, минимальная стоимость поездки из A в E сокращается с $150 (маршрут A-D-E) до $140 (маршрут A-D-C-E). Обновленная информация представлена в табл. 14.5 и на рис. 14.9.

Таблица 14.5. Шаг 4: агенты в A, B, D и C

Из города A	B	C	D	E
Шаг 1	50 (через A)	Бск	80 (через A)	Бск
Шаг 2	50 (через A)*	110 (через B)	80 (через A)	Бск
Шаг 3	50 (через A)*	100 (через D)	80 (через A)*	150 (через D)
Шаг 4	50 (через A)*	100 (через D)*	80 (через A)*	140 (через C)

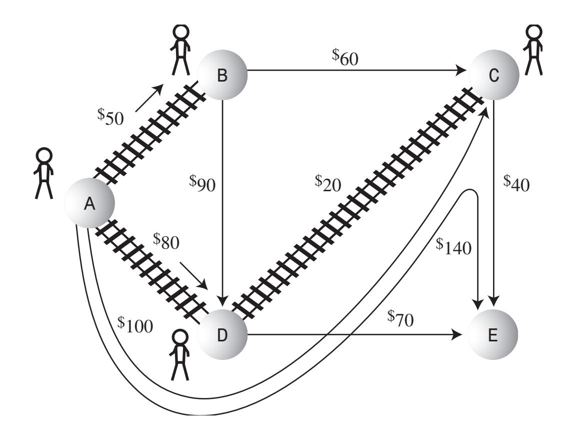


Рис. 14.9. Шаг 4 в алгоритме выбора кратчайшего пути

Последний агент: E

Самым экономичным путем из A в любой другой город, в котором пока нет своего агента, становится путь в E через D и C ($140). Вы отправляете агента в E, но агент сообщает, что из E нет ни одного маршрута в город без агентов (есть маршрут в B, но в B уже есть агент). Таблица 14.6 дополняется последней строкой; мы всего лишь добавляем звездочку в запись E, чтобы обозначить присутствие агента в этом городе.

Таблица 14.6. Шаг 4: агенты в A, B, D, C и E

Из города A	B	C	D	E
Шаг 1	50 (через A)	Бск	80 (через A)	Бск
Шаг 2	50 (через A)*	110 (через B)	80 (через A)	Бск

продолжение ⇨

Таблица 14.6 (*продолжение*)

Из города A	B	C	D	E
Шаг 3	50 (через A)*	100 (через D)	80 (через A)*	150 (через D)
Шаг 4	50 (через A)*	100 (через D)*	80 (через A)*	140 (через C)
Шаг 5	50 (через A)*	100 (через D)*	80 (через A)*	140 (через C)*

Когда в каждом городе появится свой агент, мы узнаем стоимость проезда из A в каждый из остальных городов. Работа алгоритма закончена. Без каких-либо дополнительных вычислений мы получили стоимости самых экономичных маршрутов из A во все остальные города.

Это описание демонстрирует основные принципы алгоритма Дейкстры. Ключевые моменты:

- Каждый раз при отправке агента в новый город вы используете информацию, полученную от него, для обновления списка стоимости проезда. В списке сохраняется только самый экономичный маршрут (из известных вам) от начальной точки до заданного города.
- Новый агент всегда отправляется в город с самым экономичным маршрутом от начальной точки (а не с самым дешевым сегментом от любого города, в котором уже есть агент, как при построении минимального остовного дерева).

Приложение GraphDW Workshop

За тем, как работает алгоритм Дейкстры, можно понаблюдать в приложении GraphDW Workshop. Создайте в этом приложении граф, изображенный на рис. 14.6. Результат должен выглядеть так, как показано на рис. 14.10. (О том, как вывести под графом таблицу, будет рассказано ниже.) Граф является взвешенным и направленным, поэтому при создании ребра необходимо ввести вес, а указатель мыши необходимо перетаскивать в правильном направлении, от начала к концу.

Когда построение графа будет завершено, щелкните на кнопке `Path`, а затем по запросу щелкните на вершине A. Еще несколько щелчков на кнопке `Path` включают A в дерево (вершина выделяется красным контуром).

Массив кратчайшего пути

При следующем нажатии кнопки `Path` под графом появляется таблица (рис. 14.10). Алгоритм Дейкстры начинает работу с копирования соответствующей строки матрицы смежности (то есть строки начальной вершины) в массив. (Не забудьте, что матрицу смежности можно просмотреть в любой момент при помощи кнопки `View`.) Этот массив называется «массивом кратчайших путей». Он соответствует последней строке таблицы в блокноте в нашем примере с вычислением самого экономичного маршрута. В массиве хранятся текущие версии кратчайших путей к другим вершинам, которые мы будем называть *конечными* вершинами. В таблице 14.7 конечные вершины представлены заголовками столбцов.

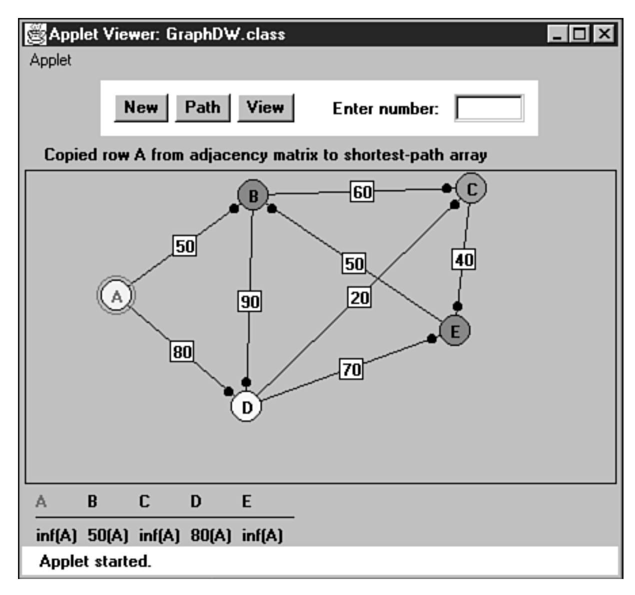


Рис. 14.10. Железнодорожный граф в приложении GraphDW

Таблица 14.7. Шаг 1: массив кратчайших путей

A	B	C	D	E
Бск (A)	50 (A)	Бск (A)	80 (A)	Бск (A)

В приложении за длинами кратчайших путей в массиве указываются *родительские* вершины в круглых скобках. Родительской называется вершина, достигнутая непосредственно перед достижением конечной вершины. В данном примере в качестве родителя везде указывается вершина A, потому что алгоритм переместился от A всего на одно ребро.

Если стоимость проезда неизвестна (или не имеет смысла, как для маршрута A-A), она представляется в виде бесконечной величины («бск» в таблице маршрутов). Обратите внимание: заголовки столбцов тех вершин, которые уже были добавлены в дерево, выделены красным цветом. Данные этих столбцов остаются неизменными.

Минимальное расстояние

Изначально алгоритм знает расстояния от A до других вершин, находящихся на расстоянии ровно одного ребра от A. Смежными с A являются только B и D, поэтому расстояния приводятся только для них. Алгоритм выбирает минимальное расстояние. При следующем нажатии кнопки `Path` появляется сообщение о том, что на минимальном расстоянии от A (50) находится вершина B. Алгоритм добавляет вершину в дерево, поэтому при следующем щелчке выводится сообщение о включении вершины B в дерево.

На графе вершина В выделяется контуром, а заголовок столбца В окрашивается в красный цвет. Ребро от А к В темнеет, что указывает на его принадлежность к дереву.

Перебор столбцов в массиве кратчайших путей

Теперь алгоритму известны не только все ребра, ведущие из А, но и все ребра из В. Он перебирает массив кратчайших путей столбец за столбцом, проверяя, нельзя ли вычислить более короткий путь на основании новой информации. Вершины, уже находящиеся в дереве (А и В), пропускаются. Сначала проверяется столбец С. Мы видим сообщение о том, что суммарный вес ребер A-B (50) и B-C (60) меньше веса ребра A-C (бесконечность).

Алгоритм обнаружил путь к С, который короче пути в массиве (бесконечность в столбце С). Вес ребра от А к В равен 50 (алгоритм находит это значение в столбце В массива кратчайших путей), а вес ребра от В к С равен 60 (находится на пересечении строки В и столбца С матрицы смежности). Сумма равна 110. Расстояние 110 меньше бесконечности, поэтому алгоритм обновляет массив кратчайших путей для столбца С и вставляет в него значение 110. За ним указывается В в круглых скобках — последняя вершина перед С; В является родителем С.

Затем анализируется столбец D. В сообщении говорится, что сумма A-B (50) и B-D (90) больше либо равна весу A-D (80). Алгоритм сравнивает ранее приводившееся расстояние от А до D, равное 80 (прямой маршрут) с потенциальным маршрутом через В (то есть A-B-D). Но суммарный вес последнего маршрута равен 140 (50 + 90). Результат больше 80, поэтому значение 80 не изменяется.

Для столбца Е выводится сообщение о том, что суммарный вес сегментов A-B (50) и B-E (бесконечность) больше либо равен весу A-E (бесконечность). Соответственно столбец Е не изменяется, а массив кратчайших путей теперь выглядит так, как показано в табл. 14.8.

Таблица 14.8. Шаг 2: массив кратчайших путей

A	B	C	D	E
Бск (A)	50 (A)	110 (B)	80 (A)	Бск (A)

Теперь проясняется роль родительской вершины, указываемой в круглых скобках после каждого расстояния. В каждом столбце содержится расстояние от А до конечной вершины. Родителем является непосредственный предшественник конечной вершины на пути от А. В столбце С родительской вершиной является В; это означает, что кратчайший путь от А к С проходит через вершину В непосредственно перед попаданием в С. Эта информация используется алгоритмом для включения в дерево соответствующей вершины. (При бесконечном расстоянии родительская вершина не имеет смысла и вместо нее выводится А.)

Новое минимальное расстояние

После обновления массива кратчайших путей алгоритм ищет кратчайшее расстояние в массиве (еще раз нажмите кнопку `Path`). Сообщение гласит, что минимальное расстояние от A равно 80 — для вершины D. Соответственно в дерево включается новая вершина D и ребро.

Снова и снова

Алгоритм снова перебирает массив кратчайших путей, проверяя и обновляя расстояния для конечных вершин, не входящих в дерево; в этой категории остаются только вершины C и E. Результат обновления столбцов C и E приведен в табл. 14.9.

Таблица 14.9. Шаг 3: массив кратчайших путей

A	B	C	D	E
Бск (A)	50 (A)	100 (D)	80 (A)	150 (D)

Кратчайший путь от A к вершине, не принадлежащей дереву, имеет вес 100 для вершины C. Соответственно вершина C включается в дерево.

При следующем просмотре массива кратчайших путей рассматривается только расстояние до E. Оно может быть сокращено прохождением через C; обновленные данные приведены в табл. 14.10.

Таблица 14.10. Шаг 3: массив кратчайших путей

A	B	C	D	E
Бск (A)	50 (A)	100 (D)	80 (A)	140 (C)

Последняя вершина E включена в дерево, а работа алгоритма закончена. В массиве кратчайших путей содержатся наименьшие расстояния от A до всех остальных вершин. Дерево состоит из вершин и ребер AB, AD, DC и CE, выделенных толстыми линиями.

Последовательность вершин по кратчайшему пути к любой вершине. Например, в кратчайшем пути к E родителем вершины E является вершина C, приведенная в массиве в круглых скобках. Родителем C (также согласно данным из массива) является вершина D, а вершине D предшествует A. Таким образом, кратчайший путь от A к E проходит по маршруту A-D-C-E.

Поэкспериментируйте с приложением GraphDW; начните с небольших графов. Через некоторое время вы поймете суть алгоритма Дейкстры и сможете предсказать, что он сделает на следующем шаге.

Реализация на языке Java

Алгоритм выбора кратчайшего пути — один из самых сложных в этой книге, но он вполне по силам простым смертным. Сначала мы рассмотрим вспомогательный класс, затем основной метод выполнения алгоритма path() и в завершение два метода, вызываемых path() для выполнения специализированных задач.

Массив sPath и класс DistPar

Как мы уже видели, основной структурой данных в алгоритме кратчайшего пути является массив с минимальными расстояниями от начальной вершины до других (конечных) вершин. В ходе выполнения алгоритма эти расстояния изменяются, а на последнем шаге элементы массива содержат фактические кратчайшие расстояния от начала. В приведенном коде этот массив называется sPath[].

Как мы уже видели, очень важно знать не только минимальное расстояние от начальной вершины до конечной, но и путь, на котором это расстояние достигается. К счастью, хранить весь путь в явном виде не обязательно. Достаточно хранить в массиве родителя конечную вершину (то есть вершину, предшествующую конечной). Если в столбце C хранится запись 100(D), это означает, что длина кратчайшего пути от A к C равна 100, и вершина D непосредственно предшествует C на этом пути.

Существует много разных способов хранения информации о родительской вершине. Мы объединили родительскую вершину с расстоянием в классе DistPar; объекты этого класса хранятся в элементах массива sPath[].

```
class DistPar             // Расстояние и родительская вершина
   {                      // Объекты сохраняются в массиве sPath
   public int distance;   // Расстояние от начальной вершины до текущей
   public int parentVert; // Текущий родитель вершины
   public DistPar(int pv, int d)  // Конструктор
      {
      distance = d;
      parentVert = pv;
      }
   }
```

Метод path()

Метод path() непосредственно реализует алгоритм выбора кратчайшего пути. В нем используется класс DistPar и класс Vertex, уже знакомый по программе mstw.java (см. листинг 14.1). Метод path() является методом класса Graph, который тоже встречался нам в несколько измененном виде в программе mstw.java.

```
public void path()                // Выбор кратчайших путей
   {
   int startTree = 0;             // Начиная с вершины 0
   vertexList[startTree].isInTree = true;
```

```
   nTree = 1;                    // Включение в дерево

   // Перемещение строки расстояний из adjMat в sPath
   for(int j=0; j<nVerts; j++)
      {
      int tempDist = adjMat[startTree][j];
      sPath[j] = new DistPar(startTree, tempDist);
      }

   // Пока все вершины не окажутся в дереве
   while(nTree < nVerts)
      {
      int indexMin = getMin();    // Получение минимума из sPath
      int minDist = sPath[indexMin].distance;

      if(minDist == INFINITY)     // Если все расстояния бесконечны
         {                        // или уже находятся в дереве,
         System.out.println("There are unreachable vertices");
         break;                   // построение sPath завершено
         }
      else
         {                        // Возврат currentVert
         currentVert = indexMin;  // к ближайшей вершине
         startToCurrent = sPath[indexMin].distance;
         // Минимальное расстояние от startTree
         // до currentVert равно startToCurrent
         }
      // Включение текущей вершины в дерево
      vertexList[currentVert].isInTree = true;
      nTree++;
      adjust_sPath();             // Обновление массива sPath[]
      }
   displayPaths();               // Вывод содержимого sPath[]

   nTree = 0;                    // Очистка дерева
   for(int j=0; j<nVerts; j++)
      vertexList[j].isInTree = false;
   }
```

Начальная вершина всегда хранится в ячейке 0 массива vertexList[]. Выполнение метода path() начинается с включения этой вершины в дерево. По ходу работы алгоритма другие вершины также будут включаться в дерево. Класс Vertex содержит флаг, обозначающий принадлежность объекта вершины к дереву. Чтобы включить вершину в дерево, необходимо установить флаг и увеличить nTree — счетчик вершин в дереве. Затем path() копирует расстояния из соответствующей строки матрицы смежности в sPath[]. Ей всегда является строка 0, потому что для простоты мы считаем, что индекс начальной вершины всегда равен 0. Изначально в поле родительской вершины всех элементов sPath[] хранится начальная вершина A.

Далее метод входит в основной цикл while алгоритма. Цикл завершается после того, как все вершины будут включены в дерево. Фактически в цикле выполняются три действия:

1. Выбор элемента sPath[] с наименьшим расстоянием.
2. Включение соответствующей вершины (заголовка столбца для этого элемента) в дерево. Вершина становится «текущей», то есть currentVert.
3. Обновление всех элементов sPath[] в соответствии с расстояниями от currentVert.

Если метод path() обнаруживает, что минимальное расстояние равно бесконечности, то он знает, что некоторые вершины недостижимы от начальной точки. Почему? Потому что не все вершины были включены в дерево (цикл while еще не завершился), но путь перехода к этим вершинам не был найден; в противном случае расстояние не было бы бесконечным.

Прежде чем возвращать управление, path() выводит окончательное содержимое sPath[] вызовом метода displayPaths(). Никаких других данных программа не выводит. Кроме того, path() обнуляет nTree и удаляет флаги isInTree из всех вершин на случай, если они будут использоваться другим алгоритмом (хотя к нашей программе это не относится).

Определение минимального расстояния методом getMin()

Чтобы найти элемент sPath[] с минимальным расстоянием, path() вызывает метод getMin(). Метод вполне тривиален — он перебирает элементы sPath[] и возвращает номер столбца (индекс массива) элемента с наименьшим расстоянием.

```
public int getMin()               // Поиск в sPath элемента
   {                              // с наименьшим расстоянием
   int minDist = INFINITY;        // Исходный высокий "минимум"
   int indexMin = 0;
   for(int j=1; j<nVerts; j++)    // Для каждой вершины
      {                           // Если она не включена в дерево
      if( !vertexList[j].isInTree &&  // и ее расстояние меньше
               sPath[j].distance < minDist )  // старого минимума
         {
            minDist = sPath[j].distance;
            indexMin = j;            // Обновление минимума
            }
      }
   return indexMin;               // Метод возвращает индекс
   }                              // элемента с наименьшим расстоянием
```

В основу алгоритма выбора кратчайшего пути можно было бы заложить приоритетную очередь, как было сделано в предыдущем разделе при поиске минимального остовного дерева. В этом случае метод getMin() был бы лишним; ребро с минимальным весом автоматически помещалось бы в начало массива. Тем не менее при использовании массива проще понять, что происходит в алгоритме.

Обновление sPath[] методом adjust_sPath()

Метод adjust_sPath() используется для обновления sPath[] новой информацией о вершинах, только что вставленных в дерево. При вызове этого метода вершина currentVert только что была вставлена в дерево, а startToCurrent содержит текущие данные из sPath[] для этой вершины. Метод adjust_sPath() проверяет каждую вершину в sPath[], последовательно перебирая вершины в переменной цикла column. Для каждого элемента sPath[] при условии, что вершина не включена в дерево, метод выполняет три операции:

1. Расстояние до текущей вершины (уже вычисленное и хранящееся в startToCurrent) суммируется с весом ребра от currentVert до вершины column. Результат сохраняется в переменной startToFringe.
2. Значение startToFringe сравнивается с текущими данными в sPath[].
3. Если значение startToFringe меньше, оно заменяет элемент в sPath[].

В этих операциях заключена суть алгоритма Дейкстры. Массив sPath[] обновляется кратчайшими расстояниями для всех вершин, известных в настоящее время. Код метода adjust_sPath():

```
public void adjust_sPath()
   {
   // Обновление данных в массиве кратчайших путей sPath
   int column = 1;               // Начальная вершина пропускается
   while(column < nVerts)        // Перебор столбцов
      {
      // Если вершина column уже включена в дерево, она пропускается
      if( vertexList[column].isInTree )
         {
         column++;
         continue;
         }
      // Вычисление расстояния для одного элемента sPath
                   // Получение ребра от currentVert к column
      int currentToFringe = adjMat[currentVert][column];
                   // Суммирование расстояний
      int startToFringe = startToCurrent + currentToFringe;
                   // Определение расстояния текущего элемента sPath
      int sPathDist = sPath[column].distance;

      // Сравнение расстояния от начальной вершины с элементом sPath
      if(startToFringe < sPathDist)   // Если меньше,
         {                            // данные sPath обновляются
         sPath[column].parentVert = currentVert;
         sPath[column].distance = startToFringe;
         }
      column++;
      }
}
```

Метод `main()` в программе `path.java` строит дерево, изображенное на рис. 14.6, и выводит для него массив кратчайших путей. Код метода выглядит так:

```
public static void main(String[] args)
   {
   Graph theGraph = new Graph();
   theGraph.addVertex('A');     // 0  (исходная вершина)
   theGraph.addVertex('B');     // 1
   theGraph.addVertex('C');     // 2
   theGraph.addVertex('D');     // 3
   theGraph.addVertex('E');     // 4

   theGraph.addEdge(0, 1, 50);  // AB 50
   theGraph.addEdge(0, 3, 80);  // AD 80
   theGraph.addEdge(1, 2, 60);  // BC 60
   theGraph.addEdge(1, 3, 90);  // BD 90
   theGraph.addEdge(2, 4, 40);  // CE 40
   theGraph.addEdge(3, 2, 20);  // DC 20
   theGraph.addEdge(3, 4, 70);  // DE 70
   theGraph.addEdge(4, 1, 50);  // EB 50

   System.out.println("Shortest paths");
   theGraph.path();             // Кратчайшие пути
   System.out.println();
   }
```

Программа выводит следующий результат:

```
A=inf(A) B=50(A) C=100(D) D=80(A) E=140(C)
```

Программа path.java

В листинге 14.2 приведен полный код программы `path.java`. Все ее компоненты были рассмотрены выше.

Листинг 14.2. Программа path.java

```
// path.java
// Выбор кратчайшего пути в направленном взвешенном графе
// Запуск программы: C>java PathApp
////////////////////////////////////////////////////////////////
class DistPar             // Расстояние и родительская вершина
   {                      // Объекты сохраняются в массиве sPath
   public int distance;   // Расстояние от начальной вершины до текущей
   public int parentVert; // Текущий родитель вершины
// -------------------------------------------------------------
   public DistPar(int pv, int d)  // Конструктор
      {
      distance = d;
      parentVert = pv;
      }
```

```
   }  // Конец класса DistPar
////////////////////////////////////////////////////////////////
class Vertex
   {
   public char label;        // Метка (например, 'A')
   public boolean isInTree;
// -------------------------------------------------------------
   public Vertex(char lab)   // Конструктор
      {
      label = lab;
      isInTree = false;
      }
// -------------------------------------------------------------
   }  // Конец класса Vertex
////////////////////////////////////////////////////////////////
class Graph
   {
   private final int MAX_VERTS = 20;
   private final int INFINITY = 1000000;
   private Vertex vertexList[]; // Список вершин
   private int adjMat[][];      // Матрица смежности
   private int nVerts;          // Текущее количество вершин
   private int nTree;           // Количество вершин в дереве
   private DistPar sPath[];     // Массив данных кратчайших путей
   private int currentVert;     // Текущая вершина
   private int startToCurrent;  // Расстояние до currentVert
// -------------------------------------------------------------
   public Graph()               // Конструктор
      {
      vertexList = new Vertex[MAX_VERTS];
                                // Матрица смежности
      adjMat = new int[MAX_VERTS][MAX_VERTS];
      nVerts = 0;
      nTree = 0;
      for(int j=0; j<MAX_VERTS; j++)     // Матрица смежности
         for(int k=0; k<MAX_VERTS; k++)  // заполняется
            adjMat[j][k] = INFINITY;     // бесконечными расстояниями
      sPath = new DistPar[MAX_VERTS];    // shortest paths
   }
// -------------------------------------------------------------
   public void addVertex(char lab)
      {
      vertexList[nVerts++] = new Vertex(lab);
      }
// -------------------------------------------------------------
   public void addEdge(int start, int end, int weight)
      {
      adjMat[start][end] = weight;  // (направленный граф)
      }
```

продолжение ⇨

Листинг 14.2 (*продолжение*)

```
// -------------------------------------------------------------
   public void path()                // Выбор кратчайших путей
      {
      int startTree = 0;             // Начиная с вершины 0
      vertexList[startTree].isInTree = true;
      nTree = 1;                     // Включение в дерево

      // Перемещение строки расстояний из adjMat в sPath
      for(int j=0; j<nVerts; j++)
         {
         int tempDist = adjMat[startTree][j];
         sPath[j] = new DistPar(startTree, tempDist);
         }

      // Пока все вершины не окажутся в дереве
      while(nTree < nVerts)
         {
         int indexMin = getMin();    // Получение минимума из sPath
         int minDist = sPath[indexMin].distance;

         if(minDist == INFINITY)     // Если все расстояния бесконечны
            {                        // или уже находятся в дереве,
            System.out.println("There are unreachable vertices");
            break;                   // построение sPath завершено
            }
         else
            {                        // Возврат currentVert
            currentVert = indexMin;  // к ближайшей вершине
            startToCurrent = sPath[indexMin].distance;
            // Минимальное расстояние от startTree
            // до currentVert равно startToCurrent
            }
         // Включение текущей вершины в дерево
         vertexList[currentVert].isInTree = true;
         nTree++;
         adjust_sPath();             // Обновление массива sPath[]
         }

      displayPaths();                // Вывод содержимого sPath[]

      nTree = 0;                     // Очистка дерева
      for(int j=0; j<nVerts; j++)
         vertexList[j].isInTree = false;
      }
// -------------------------------------------------------------
   public int getMin()               // Поиск в sPath элемента
      {                              //  с наименьшим расстоянием
      int minDist = INFINITY;        // Исходный высокий "минимум"
      int indexMin = 0;
```

```
      for(int j=1; j<nVerts; j++)    // Для каждой вершины
         {                           // Если она не включена в дерево
         if( !vertexList[j].isInTree &&  // и ее расстояние меньше
                  sPath[j].distance < minDist )  // старого минимума
            {
            minDist = sPath[j].distance;
            indexMin = j;            // Обновление минимума
            }
         }
      return indexMin;               // Метод возвращает индекс
      }                              // элемента с наименьшим расстоянием
// -------------------------------------------------------------
   public void adjust_sPath()
      {
      // Обновление данных в массиве кратчайших путей sPath
      int column = 1;                // Начальная вершина пропускается
      while(column < nVerts)         // Перебор столбцов
         {
         // Если вершина column уже включена в дерево, она пропускается
         if( vertexList[column].isInTree )
            {
            column++;
            continue;
            }
         // Вычисление расстояния для одного элемента sPath
                    // Получение ребра от currentVert к column
         int currentToFringe = adjMat[currentVert][column];
                    // Суммирование расстояний
         int startToFringe = startToCurrent + currentToFringe;
                    // Определение расстояния текущего элемента sPath
         int sPathDist = sPath[column].distance;

         // Сравнение расстояния от начальной вершины с элементом sPath
         if(startToFringe < sPathDist)   // Если меньше,
            {                            // данные sPath обновляются
            sPath[column].parentVert = currentVert;
            sPath[column].distance = startToFringe;
            }
         column++;
         }
   }
// -------------------------------------------------------------
   public void displayPaths()
      {
      for(int j=0; j<nVerts; j++) // display contents of sPath[]
         {
         System.out.print(vertexList[j].label + "="); // B=
         if(sPath[j].distance == INFINITY)
            System.out.print("inf");                  // inf
```

продолжение ⇨

Листинг 14.2 (*продолжение*)

```
        else
           System.out.print(sPath[j].distance);      // 50
        char parent = vertexList[ sPath[j].parentVert ].label;
        System.out.print("(" + parent + ") ");       // (A)
        }
      System.out.println("");
      }
// -------------------------------------------------------------
   }  // Конец класса Graph
////////////////////////////////////////////////////////////////
class PathApp
{
   public static void main(String[] args)
      {
      Graph theGraph = new Graph();
      theGraph.addVertex('A');     // 0  (исходная вершина)
      theGraph.addVertex('B');     // 1
      theGraph.addVertex('C');     // 2
      theGraph.addVertex('D');     // 3
      theGraph.addVertex('E');     // 4

      theGraph.addEdge(0, 1, 50);  // AB 50
      theGraph.addEdge(0, 3, 80);  // AD 80
      theGraph.addEdge(1, 2, 60);  // BC 60
      theGraph.addEdge(1, 3, 90);  // BD 90
      theGraph.addEdge(2, 4, 40);  // CE 40
      theGraph.addEdge(3, 2, 20);  // DC 20
      theGraph.addEdge(3, 4, 70);  // DE 70
      theGraph.addEdge(4, 1, 50);  // EB 50

      System.out.println("Shortest paths");
      theGraph.path();             // Кратчайшие пути
      System.out.println();
      }
   }  // Конец класса PathApp
////////////////////////////////////////////////////////////////
```

Поиск кратчайших путей между всеми парами вершин

Когда в главе 13 речь шла о связности графов, мы хотели узнать, возможен ли перелет от Афин до Мурманска при любом количестве пересадок. Когда вы имеете дело со взвешенным графом, появляется второй разумный вопрос: сколько будет стоить такое путешествие?

Чтобы узнать, возможно ли перемещение между вершинами, мы строили таблицу связности. Для взвешенного графа аналогичная таблица должна содержать

минимальные стоимости перемещения между всеми парами вершин по разным ребрам. Задача построения такой таблицы называется задачей *поиска кратчайших путей между всеми парами вершин*.

Чтобы построить такую таблицу, можно выполнить программу `path.java` для всех вершин попеременно. Примерный результат показан на рис. 14.11.

Таблица 14.11. Таблица кратчайших путей между всеми парами вершин

	A	B	C	D	E
A	–	50	100	80	140
B	–	–	60	90	100
C	–	90	–	180	40
D	–	110	20	–	60
E	–	50	110	140	–

В предыдущей главе было показано, что для создания таблицы с информацией о достижимости вершин от конкретной вершины (за один или несколько шагов) существует более быстрый способ — алгоритм Уоршелла. Аналогичное решение для взвешенных графов использует алгоритм Флойда, предложенный Робертом Флойдом в 1962 году. Это другой способ построения данных из табл. 14.11.

Рассмотрим работу алгоритма Флойда на примере упрощенного графа. На рис. 14.11 изображен взвешенный направленный граф с матрицей смежности.

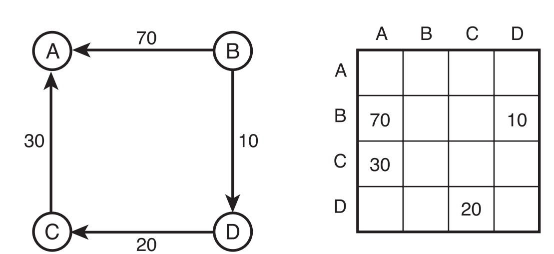


Рис. 14.11. Взвешенный граф и его матрица смежности

В матрице смежности указана стоимость всех путей, состоящих из одного ребра. Мы хотим расширить матрицу таким образом, чтобы в ней указывалась стоимость всех путей независимо от их длины. Например, из рис. 14.11 видно, что стоимость перехода от B к C равна 30 (10 от B к D плюс 20 от D к C).

Как и в алгоритме Уоршелла, мы будем систематически изменять матрицу смежности. Алгоритм последовательно проверяет каждую ячейку каждой строки. Если вес отличен от нуля (допустим, 30 для строки C и столбца A), мы просматриваем содержимое столбца C (потому что значение 30 находится в строке C). При обнаружении данных в столбце C (40 на пересечении со строкой D) мы знаем, что от C к A ведет путь с весом 30, а от D к C — путь с весом 40. Из этого можно сделать вывод, что от D к A существует двухшаговый путь с суммарным весом 70. На рис. 14.12 показан результат применения алгоритма Флойда к графу на рис. 14.11.

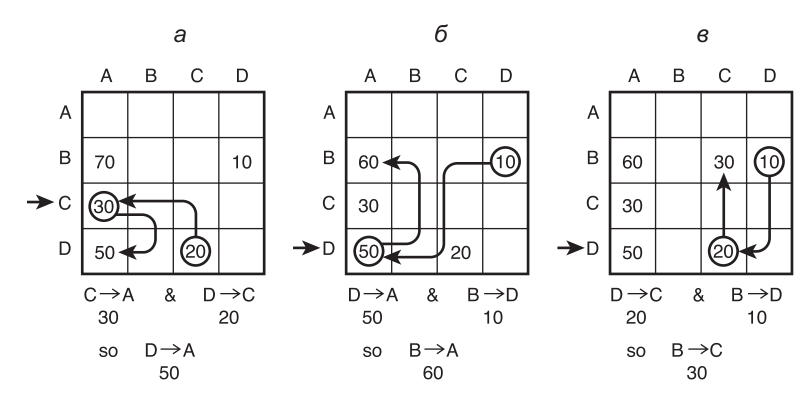


Рис. 14.12. Алгоритм Флойда: *а*) $y = 2$, $x = 0$, $z = 3$; *б*) $y = 3$, $x = 0$, $z = 1$; *в*) $y = 3$, $x = 2$, $z = 1$

Строка A пуста, делать здесь нечего. В строке B столбец A содержит значение 70, а столбец D — значение 10, но в столбце B данных нет, поэтому элементы строки B не удастся объединить с ребрами, заканчивающимися в B.

Однако в строке C мы находим значение 30 на пересечении со столбцом A. При обращении к столбцу C в строке D обнаруживается значение 20. Теперь нам известны пути от C к A (вес 30) и от D к C с весом 20; следовательно, от D к A существует путь с весом 50.

В строке D возникает интересная ситуация со снижением текущей стоимости. Столбец A содержит значение 50. Кроме того, на пересечении строки B и столбца D находится значение 10; следовательно, от B к A ведет путь со стоимостью 60. Однако в этой ячейке уже хранится значение 70. Так как 60 меньше 70, мы заменяем 70 новым значением 60. При наличии нескольких путей между вершинами в таблице должна храниться стоимость пути с наименьшим суммарным весом.

Реализация алгоритма Флойда сходна с реализацией алгоритма Уоршелла, но вместо того, чтобы просто вставлять единицу в таблицу при обнаружении пути из двух ребер, мы суммируем веса двух одношаговых путей и вставляем сумму. Подробности реализации предоставляются читателю для самостоятельной работы.

Эффективность

До настоящего момента мы не рассматривали эффективность различных алгоритмов, работающих с графами. Ситуация усложняется наличием двух способов представления графов: матрицы смежности и списков смежности.

При использовании матрицы смежности описанные алгоритмы в основном выполняются со сложностью $O(V^2)$, где V — количество вершин. Почему? Анализ алгоритмов показывает, что они основаны на однократной проверке каждой вершины, после чего для каждой вершины просматривается соответствующая строка матрицы смежности. Иначе говоря, просматривается каждая ячейка матрицы смежности, состоящей из V^2 ячеек.

Для больших матриц сложность $O(V^2)$ оставляет желать лучшего. Для насыщенных графов возможности улучшения эффективности ограничены. (*Насыщенным* называется граф с большим количеством ребер, для которого заполнено большинство ячеек матрицы смежности.)

Однако большинство графов относится к категории разреженных. Не существует четкого определения того, при каком количестве ребер граф может считаться насыщенным или разреженным, но если каждая вершина большого графа соединяется с соседними вершинами всего несколькими ребрами, такой граф обычно относится к разреженным.

В разреженном графе время работы алгоритма обычно улучшается при переходе от представления в виде матрицы смежности к спискам смежности. Причины понятны: алгоритм не тратит время на проверку ячеек матрицы смежности, не содержащих ребер.

Для невзвешенных графов алгоритм обхода в глубину со списками смежности выполняется за время $O(V + E)$, где V — количество вершин, а E — количество ребер. Для взвешенных графов алгоритм построения минимального остовного дерева и алгоритм кратчайшего пути выполняются за время $O((E + V) \log V)$. В больших разреженных графах эти показатели могут стать значительным улучшением по сравнению с матрицей смежности. С другой стороны, реализация алгоритмов несколько усложняется, поэтому в этой главе использовалось исключительно представление с матрицей смежности. За примерами реализации алгоритмов графов на основе списков смежности обращайтесь к Седжвику (см. приложение Б) и другим источникам.

Алгоритмы Уоршелла и Флойда работают медленнее других алгоритмов, представленных в книге. Оба алгоритма требуют времени $O(V^3)$, так как в их реализации используется тройной вложенный цикл.

Неразрешимые задачи

В книге нам встречались алгоритмы с разной сложностью: от $O(1)$ к $O(N)$, $O(N^*\log N)$, $O(N^2)$ и (для алгоритмов Уоршелла и Флойда) $O(N^3)$. Даже при сложности $O(N^3)$ при тысячных значениях N вычисления могут быть выполнены за приемлемое время. Алгоритмы с такой сложностью могут использоваться для поиска решений большинства реальных задач.

С другой стороны, у некоторых алгоритмов сложность оказывается настолько большой, что они могут применяться только для относительно небольших значений N. Многие реальные задачи, требующие применения таких алгоритмов, просто не могут быть решены за сколько-нибудь разумный промежуток времени. Такие задачи называются неразрешимыми. Также встречается термин «НП-полные задачи», где сокращение «НП» означает «недетерминированный полиномиальный» (к сожалению, подробные объяснения выходят за рамки книги).

Обход доски ходом шахматного коня

Задача обхода доски ходом шахматного коня (программный проект 13.5 в главе 13) относится к категории неразрешимых из-за слишком большого количества возможных ходов. Общее количество возможных последовательностей ходов плохо поддается точным вычислениям, но приблизительная оценка возможна. Максимальное количество полей, на которые может быть сделан ход с каждого поля, равно 8. Это число сокращается за счет ходов, ведущих за край доски, а также ходов на поля, которые уже были посещены ранее. На ранней стадии обхода количество возможных ходов будет оставаться близким к 8, но будет постепенно сокращаться по мере заполнения доски. Предположим (более чем консервативно), что из каждой позиции возможно в среднем только два хода. После первого хода конь может посетить еще 63 оставшихся поля. Таким образом, общее количество возможных ходов равно 2^{63}, или около 10^{19}. Компьютер обрабатывает до миллиона ходов в секунду, или 10^6. Год состоит примерно из 10^7 секунд, поэтому за год компьютер обработает 10^{13} ходов. Таким образом, решение задачи методом «грубой силы» займет 10^6 (около миллиона) лет.

Разрешимость этой конкретной задачи можно несколько улучшить за счет применения стратегий «усечения» игрового дерева. Одна из них — эвристическое правило Уорнсдорфа (Г. К. фон Уорнсдорф, 1823) — указывает, что перемещение всегда должно вести к полю с минимальным количеством возможных выходов.

Задача коммивояжера

Другая известная неразрешимая задача: представьте, что вы — коммивояжер и вам нужно объехать все города, в которых у вас имеются клиенты. Вам хотелось бы свести к минимуму пройденное расстояние. Известно расстояние между любыми двумя городами. Вы хотите начать поездку со своего города, посетить каждый город с клиентом ровно один раз и вернуться в исходный город. В какой последовательности следует посещать города, чтобы свести к минимуму пройденное расстояние? В теории графов эта задача называется *задачей коммивояжера*.

На рис. 14.13 представлено расположение городов и расстояния между ними. Как будет выглядеть кратчайший путь перемещения от A по всем городам и обратно в A? Обратите внимание: в задаче не требуется, чтобы каждая пара городов была соединена ребром.

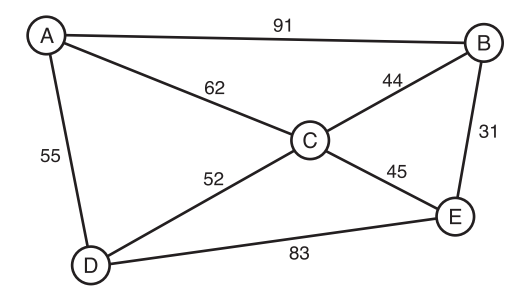


Рис. 14.13. Города и расстояния

Чтобы найти кратчайший маршрут, следует составить список всех возможных перестановок городов (A-B-C, B-C-A, C-B-A и т. д.), а также вычислить общее расстояние для каждой перестановки. Общая продолжительность маршрута ABCEDA равна 318. Маршрут ABCDEA невозможен из-за отсутствия ребра из E в A.

К сожалению, количество перестановок может быть очень большим: оно вычисляется как факториал количества городов (не считая исходного). Если требуется посетить 6 городов, то первый город выбирается из 6 вариантов, второй — из 5 вариантов, третий — из 4 и т. д; итого 6*5*4*3*2*1, или 720 возможных маршрутов. Даже для 50 городов решить задачу простым перебором нереально. Некоторые стратегии позволяют сократить количество проверяемых последовательностей, но они помогают лишь незначительно. Задача реализуется с использованием взвешенного графа: веса представляют расстояния между городами, а вершины — сами города. Граф может быть ненаправленным, если расстояние от A до B совпадает с расстоянием от B до A, как это обычно бывает при переездах. Если веса представляют стоимость авиабилетов, они могут изменяться в зависимости от направления; в таком случае используется направленный граф.

Гамильтоновы циклы

Задача, отчасти похожая на задачу коммивояжера, но более абстрактная — задача поиска гамильтонового цикла графа. Как упоминалось ранее, циклом называется путь, который начинается и заканчивается в одной вершине. Гамильтонов цикл посещает каждую вершину графа ровно один раз. В отличие от задачи коммивояжера, расстояния нас не интересуют; нужно лишь определить, существует ли такой цикл. На рис. 14.13 маршрут ABCEDA является гамильтоновым циклом, а маршрут ABCDEA — не является. Задача обхода доски ходом шахматного коня тоже является примером гамильтонова цикла (если предположить, что конь возвращается на исходную клетку).

Поиск гамильтонова цикла выполняется за то же время $O(N!)$, что и решение задачи коммивояжера. Высокая сложность в O-синтаксисе — например, 2^N и $N!$, возрастающая еще быстрее; 2^N — часто называется *экспоненциальной*.

Итоги

- Во взвешенном графе ребрам ставятся в соответствие числа, называемые весами. Они могут представлять расстояния, затраты, время и другие величины.
- Минимальное остовное дерево во взвешенном графе минимизирует вес ребер, необходимых для соединения всех вершин.
- Для построения минимального остовного дерева взвешенного графа может использоваться алгоритм на базе приоритетной очереди.
- Минимальное остовное дерево взвешенного графа моделирует реальные ситуации, например прокладку кабелей между городами.

- Задача выбора кратчайшего пути в невзвешенном графе заключается в определении минимального количества ребер между двумя вершинами.
- Для взвешенных графов при решении задачи выбора кратчайшего пути строится путь с минимальным суммарным весом ребер.
- Задача выбора кратчайшего пути для взвешенных графов может решаться при помощи алгоритма Дейкстры.
- Для больших разреженных графов алгоритмы обычно работают быстрее при представлении графа в формате списка смежности (вместо матрицы смежности).
- В задаче построения кратчайших путей для всех пар вершин определяется суммарный вес ребер между всеми парами вершин в графе. Для решения этой задачи может использоваться алгоритм Флойда.
- Некоторые алгоритмы выполняются за экспоненциальное время. Такие алгоритмы могут применяться лишь к графам с относительно небольшим количеством вершин.

Вопросы

Следующие вопросы помогут читателю проверить качество усвоения материала. Ответы на них приведены в приложении В.

1. Веса во взвешенном графе являются свойством ________ графа.
2. Во взвешенном графе минимальное остовное дерево стремится свести к минимуму:
   a) количество ребер от начальной до заданной вершины;
   b) количество ребер, соединяющих все вершины;
   c) суммарный вес ребер от начальной до заданной вершины;
   d) суммарный вес ребер, соединяющих все вершины.
3. Вес минимального остовного дерева зависит от выбора начальной вершины (Да/Нет).
4. Что удаляется из приоритетной очереди в алгоритме построения минимального остовного дерева?
5. В примере с кабельной сетью каждое ребро, добавляемое в минимальное остовное дерево, соединяет:
   a) начальную вершину со смежной вершиной;
   b) уже подключенный город с неподключенным городом;
   c) текущую вершину со смежной вершиной;
   d) два города с офисами.
6. Алгоритм построения минимального остовного дерева исключает ребро из списка, если оно ведет к вершине, которая __________ .

7. Задача выбора кратчайшего пути имеет смысл только для направленных графов (Да/Нет).
8. Алгоритм Дейкстры находит кратчайший путь:
   a) от одной заданной вершины до всех остальных вершин;
   b) от одной заданной вершины до другой заданной вершины;
   c) от всех вершин до всех остальных вершин, до которых можно добраться через одно ребро;
   d) от всех вершин до всех остальных вершин, до которых можно добраться через несколько ребер.
9. Правило алгоритма Дейкстры требует, чтобы в дерево всегда включалась вершина, ближайшая к начальной вершине (Да/Нет).
10. В примере с железнодорожными билетами периферийным является город:
    a) расстояние до которого известно, но расстояния от которого неизвестны;
    b) который включен в дерево;
    c) расстояние до которого известно и который включен в дерево;
    d) о котором ничего не известно.
11. В задаче построения кратчайших путей для всех пар вершин ищется кратчайший путь:
    a) от начальной вершины до всех остальных вершин;
    b) от каждой вершины до всех остальных вершин;
    c) от начальной вершины до всех остальных вершин, находящихся от нее на расстоянии одного ребра;
    d) от каждой вершины до всех остальных вершин, находящихся от нее на расстоянии одного и более ребра.
12. Алгоритм Флойда для взвешенных графов является тем, чем ________ является для невзвешенных графов.
13. Алгоритм Флойда использует представление графа в виде _________ .
14. За какое приблизительно время (в O-синтаксисе) решается задача обхода доски ходом шахматного коня?
15. Является ли маршрут ABCEDA на рис. 14.13 оптимальным решением задачи коммивояжера?

Упражнения

Упражнения помогут вам глубже усвоить материал этой главы. Программирования они не требуют.

1. Используйте приложение GraphW Workshop для построения минимального остовного дерева графа на рис. 14.6. Граф следует считать ненаправленным (то есть на стрелки не обращайте внимания).

2. Решите задачу выбора кратчайшего пути для графа на рис. 14.6 в приложении GraphDW Workshop, но вычислите для всех ребер новые веса, вычитая приведенные на рисунке веса из 100.
3. Нарисуйте граф из пяти вершин и пяти ребер. Реализуйте алгоритм Дейкстры для этого графа, используя карандаш и бумагу. На каждом шаге записывайте текущее состояние дерева и массива кратчайших путей.

Программные проекты

В этом разделе читателю предлагаются программные проекты, которые укрепляют понимание материала и демонстрируют практическое применение концепций этой главы.

14.1. Измените программу `path.java` (см. листинг 14.2), чтобы она выводила таблицу минимальной стоимости перемещения от любой вершины до всех остальных вершин. Для этого придется внести изменения в методы, предполагающие, что начальной вершиной всегда является вершина A.

14.2. Мы рассмотрели варианты представления графов в виде матрицы смежности или списков смежности. Также для представления графа могут использоваться сслыки Java: в объекте `Vertex` хранится список ссылок на другие вершины, с которыми он соединен. Такое использование ссылок выглядит особенно логично в направленных графах, потому что ссылки «указывают» из одной вершины на другую. Напишите программу, реализующую эту схему. Метод `main()` пишется по аналогии с методом `main()` программы `path.java` (см. листинг 14.2): он тоже должен строить граф, изображенный на рис. 14.6, с использованием тех же вызовов `addVertex()` и `addEdge()`. Далее метод должен выводить таблицу связности графа, чтобы доказать, что граф построен верно. Вес каждого ребра необходимо где-то хранить. Одно из возможных решений — класс `Edge` с полями для веса ребра и вершины, на которой оно кончается. Для каждой вершины ведется список объектов `Edge` (то есть ребер, начинающихся от этой вершины).

14.3. Реализуйте алгоритм Флойда. Начните с программы `path.java` (см. листинг 14.2) и внесите в нее необходимые изменения. В частности, из программы можно удалить весь код поиска кратчайших путей. Недостижимые вершины, как и прежде, должны представляться бесконечностью. Это позволит вам обойтись без проверки на нуль при сравнении текущего веса с вычисленным. Веса всех возможных маршрутов должны быть меньше бесконечности. Метод `main()` должен поддерживать возможность графов любой сложности.

14.4. Реализуйте задачу коммивояжера (см. раздел «Неразрешимые задачи» этой главы). Несмотря на неразрешимость, вам удастся без особых проблем решить задачу для небольших значений *N*, например 10 и менее городов. Попробуйте использовать ненаправленный граф. Переберите все возможные последовательности городов методом «грубой силы». Для генерирования перестановок воспользуйтесь программой `anagram.java` (см. листинг 6.2) из главы 6, «Рекурсия». Для представления несуществующих ребер используйте бесконечные веса. В этом случае вам не

придется прерывать обработку последовательности с несуществующими ребрами; любой суммарный вес, превышающий бесконечность, обозначает невозможный маршрут. Также не беспокойтесь об исключении симметричных маршрутов (например, включайте в выходные данные и ABCDEA, и AEDCBA).

14.5. Напишите программу, которая находит и выводит все гамильтоновы циклы взвешенного ненаправленного графа.

Глава 15
Рекомендации по использованию

В этой главе приводится краткая сводка того, что мы узнали к настоящему моменту. Особое внимание уделяется выбору структуры данных или алгоритма для конкретной ситуации.

К рекомендациям, приведенным в этой главе, следует относиться осмотрительно. Материал неизбежно имеет крайне обобщенный характер. Все реальные ситуации не похожи друг на друга, поэтому может оказаться, что приведенное описание не подходит для вашей конкретной задачи. Глава условно разделена на несколько тем:

- Структуры данных общего назначения: массивы, связанные списки, деревья, хеш-таблицы.
- Специализированные структуры данных: стеки, очереди, приоритетные очереди, графы.
- Сортировка: метод вставки, сортировка Шелла, быстрая сортировка, сортировка слиянием, пирамидальная сортировка.
- Графы: матрица смежности, список смежности.
- Внешнее хранение данных: последовательное хранение, индексированные файлы, В-деревья, хеширование.

ПРИМЕЧАНИЕ

За подробной информацией об этих темах обращайтесь к соответствующим главам книги.

Структуры данных общего назначения

Если задача требует хранения реальных данных: сведений о работниках, складской информации, списков контактов, данных о продажах и т. д., вам понадобится структура данных общего назначения. В книге был рассмотрен целый ряд структур данных этой категории: массивы, связанные списки, деревья и хеш-таблицы. Мы называем эти структуры данных «структурами общего назначения», потому что они используются для хранения и выборки данных по значениям ключей. Такие структуры ориентированы на выполнение обобщенных операций с базами данных

(в отличие от специализированных структур вроде стеков, ограничивающим доступ к элементам данных).

Какую из структур данных общего назначения следует выбрать для конкретной задачи? В первом приближении можно воспользоваться рис. 15.1, однако кроме факторов, показанных на рисунке, существует много других. Чтобы картина получилась более подробной, мы сначала исследуем некоторые общие вопросы, а затем более подробно рассмотрим отдельные структуры.

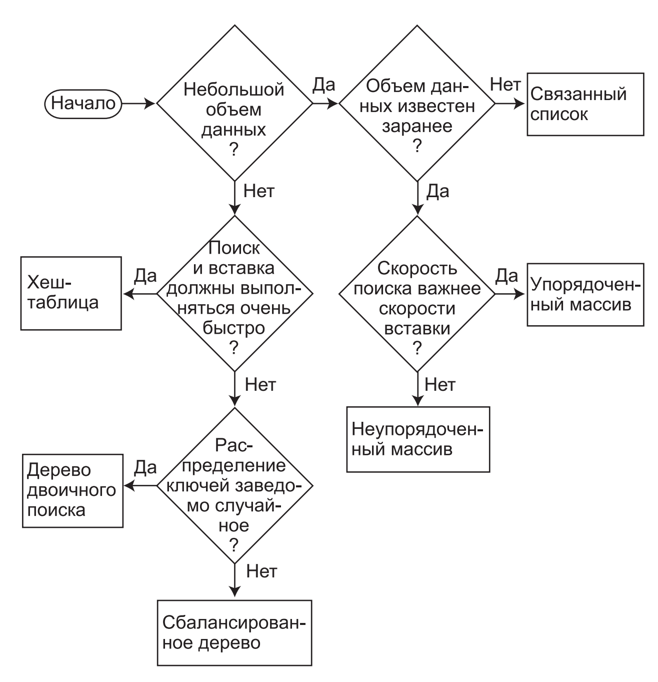


Рис. 15.1. Выбор структуры данных общего назначения

Скорость и алгоритмы

Структуры данных общего назначения можно приблизительно классифицировать по скорости: массивы и связанные списки считаются медленными, деревья — относительно быстрыми, а хеш-таблицы — очень быстрыми. Однако из табл. 15.1 вовсе не следует, что предпочтение всегда отдается более быстрым структурам. У них имеются свои оборотные стороны. Во-первых, они — в той или иной степени — усложняют реализацию по сравнению с массивами и связанными списками. Кроме того, для использования хеш-таблиц объем данных должен быть известен заранее, к тому же хеш-таблицы неэффективно используют память. Эффективность обычных двоичных деревьев для упорядоченных данных падает до $O(N)$, а сбалансированные деревья, решающие эту проблему, усложняют реализацию.

Скорость процессора

У быстрых структур данных есть свои оборотные стороны, а в пользу более медленных структур существует еще один довод. Ежегодно вычислительная мощь процессоров и скорость доступа к памяти современных компьютеров растут. Закон Мура (сформулированный Гордоном Муром в 1965 году) гласит, что производительность процессоров удваивается за каждые 18 месяцев. В результате современные компьютеры разительно отличаются по производительности от ранних моделей, и нет причин полагать, что этот рост в обозримом будущем замедлится.

Допустим, несколько лет назад компьютер за приемлемое время обрабатывал массив из 100 объектов. В наши дни компьютеры работают намного быстрее, и обработка массива из 10 000 объектов может занять столько же времени. Многие авторы приводят оценку максимального размера структуры данных, при превышении которого она начинает работать слишком медленно. Не доверяйте таким оценкам (в том числе и приведенным в этой книге). Вчерашние оценки сегодня недействительны.

Начните с простых структур данных. Реализуйте простую версию массива или связанного списка и посмотрите, что будет (если у вас нет полной уверенности в том, что их скорость будет слишком низкой для вашей задачи). Если программа выполняется за приемлемое время, остановитесь на этом. Зачем тратить время на сбалансированное дерево, если его можно заменить массивом, и это никто не заметит? Даже если ваша программа должна работать с тысячами или десятками тысяч элементов, все равно стоит посмотреть, не удастся ли обойтись массивом или связанным списком. Только если тестирование покажет, что быстродействия этих структур недостаточно, следует переходить к более сложным структурам данных.

Преимущества ссылок Java

По скорости работы с объектами Java превосходит некоторые другие языки, потому что в большинстве структур данных Java ограничивается хранением ссылок, а не самих объектов. Соответственно многие алгоритмы работают быстрее, чем в языках, в которых в структуре данных хранятся фактические объекты. При анализе алгоритмов важно не то, когда и как сохраняются сами объекты, а время «перемещения» объекта. Так как в Java перемещается только ссылка, размер объекта не имеет значения.

Конечно, в других языках — таких, как C++ — вместо объектов в полях могут храниться указатели на объекты. Это обеспечивает такой же выигрыш по быстродействию, как при использовании ссылок, но синтаксис несколько усложняется.

Библиотеки

Коммерческие библиотеки структур данных существуют для всех основных языков программирования. В некоторых языках предусмотрена встроенная поддержка структур, например классы `Vector`, `Stack` и `Hashtable` для языка Java. В C++ входит

библиотека STL (Standard Template Library) с классами для многих структур данных и алгоритмов.

Коммерческие библиотеки избавляют (полностью или частично) программиста от необходимости самостоятельно создавать структуры данных, описанные в книге. В подобных ситуациях сложные структуры данных (например, сбалансированное дерево) или алгоритмы (например, быстрая сортировка) начинают выглядеть более привлекательно. Однако вы должны убедиться в том, что класс хорошо адаптируется к вашей конкретной задаче.

Массивы

В большинстве случаев выбор структуры данных для хранения и обработки данных следует начинать с массива. Массивы особенно полезны, когда:

- объем данных относительно невелик;
- объем данных известен заранее.

При достаточно большом объеме памяти второе условие не так существенно; просто выделите массив достаточно большого размера, чтобы он справился с возможными «пиками».

Если важна скорость вставки, используйте неупорядоченный массив. Если важна скорость поиска, используйте упорядоченный массив с двоичным поиском. Удаление в массивах всегда выполняется медленно, потому что для заполнения освободившейся ячейки приходится сдвигать в среднем половину элементов. Перебор быстро выполняется в упорядоченном массиве, но не поддерживается в неупорядоченном.

Вектор (класс `Vector` в языке Java) представляет собой массив, автоматически расширяемый при заполнении. Векторы могут использоваться тогда, когда объем данных неизвестен заранее, однако их расширение может сопровождаться значительной задержкой при копировании старых данных в новое пространство.

Связанные списки

Связанные списки обычно применяются в ситуациях, когда объем данных неизвестен заранее или же с данными часто выполняются операции вставки и удаления. Связанный список автоматически расширяется при добавлении новых элементов, поэтому он может заполнить всю доступную память; в отличие от массивов он не требует заполнения «дыр», оставшихся при удалении.

Вставка в неупорядоченном списке выполняется быстро. Операции поиска и удаления относительно медленны (хотя удаление выполняется быстрее, чем в массивах), поэтому связанные списки, как и массивы, желательно использовать при сравнительно небольшом объеме данных.

Связанные списки реализуются несколько сложнее, чем массивы, но проще деревьев или хеш-таблиц.

Деревья двоичного поиска

Двоичное дерево — первая структура, которую следует рассмотреть, если массивы и связанные списки оказались слишком медленными. Дерево обеспечивает быстрое ($O(\log N)$) выполнение вставки, поиска и удаления. Перебор выполняется за время $O(N)$, максимальное для любых структур данных (по определению приходится перебирать все элементы). Также деревья позволяют быстро найти минимум и максимум и выполнить перебор диапазона элементов.

Несбалансированное двоичное дерево намного проще программируется, чем сбалансированное, но, к сожалению, вставка упорядоченных данных может сократить его быстродействие до $O(N)$ — не лучше, чем у связанного списка. Однако если вы твердо уверены в том, что данные будут поступать в случайном порядке, нет никаких причин использовать сбалансированное дерево.

Сбалансированные деревья

Из всех разновидностей сбалансированных деревьев мы рассмотрели красно-черные деревья и деревья 2-3-4. Являясь сбалансированными деревьями, они гарантируют быстродействие $O(\log N)$ независимо от того, упорядочены входные данные или нет. Однако реализация сбалансированных деревьев относительно сложна (особенно для красно-черных). Кроме того, могут возникнуть проблемы с лишними затратами памяти, что иногда бывает существенно.

Для решения проблем со сложностями программирования деревьев можно воспользоваться готовым коммерческим классом. В некоторых случаях вместо сбалансированного дерева лучше воспользоваться хеш-таблицей. Быстродействие хеш-таблиц не ухудшается при упорядоченных данных.

Также существуют другие разновидности сбалансированных деревьев: деревья AVL, расширяющиеся (splay) деревья и т. д., но на практике они используются реже, чем красно-черные деревья.

Хеш-таблицы

Хеш-таблицы обладают самой высокой скоростью среди всех структур данных, поэтому они практически всегда используются в тех случаях, когда с данными работает не человек, а компьютерная программа. Хеш-таблицы обычно используются в системах проверки орфографии и при работе с таблицами символических имен компиляторов, когда программе приходится проверять тысячи слов или символьных последовательностей за долю секунды.

Хеш-таблицы полезны и тогда, когда с данными работает человек, а не компьютер. Как упоминалось ранее, хеш-таблицы не чувствительны к упорядоченности вставляемых данных, поэтому они могут использоваться вместо сбалансированных деревьев. По простоте программирования они значительно превосходят сбалансированные деревья.

Использование хеш-таблиц сопряжено с дополнительными затратами памяти, особенно при открытой адресации. Также объем хранимых данных должен быть

относительно точно известен заранее, потому что в качестве базовой структуры данных используется массив.

Самой устойчивой реализацией является хеш-таблица, использующая метод цепочек. Исключение составляет ситуация, при которой объем данных точно известен заранее; в этом случае открытая адресация, не требующая дополнительного класса связанного списка, упрощает программирование.

Хеш-таблицы не поддерживают ни упорядоченного перебора, ни выборки минимального/максимального элемента. Если эти возможности важны для вас, лучше остановиться на дереве двоичного поиска.

Быстродействие структур данных общего назначения

В таблице 15.1 приведена сводка быстродействия основных операций для различных структур данных общего назначения.

Таблица 15.1. Структуры данных общего назначения

Структура данных	Поиск	Вставка	Удаление	Перебор
Массив	$O(N)$	$O(1)$	$O(N)$	–
Упорядоченный массив	$O(\log N)$	$O(\mathrm{N})$	$O(N)$	$O(N)$
Связанный список	$O(N)$	$O(1)$	$O(N)$	–
Упорядоченный связанный список	$O(N)$	$O(N)$	$O(N)$	$O(N)$
Двоичное дерево (в среднем)	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(N)$
Двоичное дерево (худший случай)	$O(N)$	$O(N)$	$O(N)$	$O(N)$
Сбалансированное дерево (средний и худший случай)	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(N)$
Хеш-таблица	$O(1)$	$O(1)$	$O(1)$	–

Предполагается, что вставка в неупорядоченном массиве выполняется в конце. Упорядоченный массив использует быстрый двоичный поиск, но операции вставки и удаления замедляются необходимостью перемещения в среднем половины элементов. Под перебором подразумевается посещение элементов данных в порядке возрастания или убывания ключей; прочерк означает, что операция не поддерживается.

Специализированные структуры данных

К специализированным структурам данных, рассмотренным в книге, относятся стек, очередь и приоритетная очередь. Эти структуры обычно не используются для ведения баз данных, ориентированных на взаимодействие с пользователем. Они применяются в компьютерных программах для реализации определенных алго-

ритмов. В книге встречается немало примеров такого рода — например, в главе 13, «Графы», и главе 14, «Взвешенные графы», стеки, очереди и приоритетные очереди используются для работы с графами.

Стеки, очереди и приоритетные очереди являются абстрактными типами данных (ADT), которые реализуются на базе более традиционных структур: массивов, связанных списков или (в случае приоритетных очередей) пирамид. Абстрактные типы данных предоставляют простой интерфейс для пользователя, как правило, с возможностью обращения или удаления только одного элемента данных:

- Стек: последний вставленный элемент.
- Очередь: первый вставленный элемент.
- Приоритетная очередь: элемент с наивысшим приоритетом.

Абстрактные типы данных могут рассматриваться как концептуальные вспомогательные средства. Их функциональность может быть реализована непосредственно на уровне базовой структуры данных (например, массива), но сокращенный интерфейс упрощает решение многих задач.

Рассмотренные в книге абстрактные типы не обладают удобными средствами перебора или поиска элемента по ключу.

Стек

Стек используется тогда, когда операции выполняются только с последним вставленным элементом. Таким образом, стек работает по принципу LIFO (Last In, First Out).

Стеки часто реализуются на базе массива или связанного списка. Реализация на базе массива эффективна, потому что элементы вставляются в конец массива, где их удобно удалять. Переполнение стека возможно, но маловероятно при разумном выборе размера массива — стеки редко используются для хранения огромных объемов данных.

Если стек содержит много данных, объем которых невозможно точно предсказать заранее (как, например, при стековой реализации рекурсии), в качестве базовой структуры лучше выбрать связанный список. Он работает достаточно эффективно, так как элементы быстро вставляются и удаляются в начале списка, а переполнение невозможно (если не считать возможности заполнения всей свободной памяти). Связанный список работает чуть медленнее массива, потому что при создании нового элемента для вставки необходимо выделить блок памяти, а в какой-то момент после удаления элемента из списка эта память должна быть освобождена.

Очередь

Очередь используется тогда, когда операции выполняются только с первым вставленным элементом. Таким образом, очередь работает по принципу FIFO (First In, First Out).

Очередь, как и стек, может быть реализована на базе массива или связанного списка. Обе реализации эффективны. Массив потребует дополнительного программирования циклического перехода при достижении конца массива. Связанный список может быть двусторонним, чтобы вставка и удаление выполнялись с разных концов.

Как и в случае со стеками, выбор базы реализации (массив или связанный список) определяется возможностью прогнозирования объема данных. Если объем данных известен заранее, используйте массив, а если нет — связанный список.

Приоритетная очередь

Стек используется тогда, когда операции выполняются только с элементом, обладающим наивысшим приоритетом, то есть элементом с наибольшим (а в некоторых случаях наименьшим) ключом.

Приоритетная очередь может быть реализована на базе упорядоченного массива или пирамиды. Вставка с упорядоченным массивом выполняется медленно, но зато удаление проходит быстро. В реализации на базе пирамиды и вставка, и удаление выполняются за время $O(\log N)$.

Если скорость вставки не имеет особого значения, используйте массив или двусторонний связанный список. Массив хорошо подходит при заранее известном объеме данных, а связанный список — если объем данных заранее неизвестен. Если скорость вставки важна, то предпочтение отдается пирамиде.

Таблица 15.2. Специализированные структуры данных

Структура данных	Вставка	Удаление	Комментарий
Стек (массив или связанный список)	$O(1)$	$O(1)$	Извлекается элемент, вставленный последним
Очередь (массив или связанный список)	$O(1)$	$O(1)$	Извлекается элемент, вставленный первым
Приоритетная очередь (упорядоченный массив)	$O(N)$	$O(1)$	Извлекается элемент с наивысшим приоритетом
Приоритетная очередь (пирамида)	$O(\log N)$	$O(\log N)$	Извлекается элемент с наивысшим приоритетом

Сортировка

Как и при выборе структуры данных, начать стоит с медленной, но простой сортировки, например сортировки методом вставки. Может оказаться, что вычислительная мощь современных компьютеров позволит отсортировать данные за разумное время. (Можно очень условно сказать, что медленная сортировка подойдет для количества элементов менее 1000.)

Сортировка методом вставки также хорошо подходит для приблизительно отсортированных файлов; если количество неупорядоченных элементов относительно

невелико, она выполняется за время $O(N)$. Чаще всего подобная ситуация встречается при добавлении нескольких новых элементов в уже отсортированный файл.

Если скорость сортировки методом вставки оказывается неудовлетворительной, то следующим кандидатом является сортировка Шелла. Она относительно легко реализуется и не предъявляет никаких особых требований к данным. По оценке Седжвика, она хорошо работает для 5000 и менее элементов.

Только если и сортировка Шелла оказывается слишком медленной, следует переходить к более сложным и более быстрым методам сортировки: сортировке слиянием, пирамидальной и быстрой сортировке. Сортировка слиянием требует дополнительной памяти, для пирамидальной сортировки необходима дополнительная структура данных, и оба способа по скорости уступают быстрой сортировке, поэтому при выполнении сортировки за минимальное время обычно выбирается быстрая сортировка. Но если данные недостаточно случайны, быстродействие быстрой сортировки может ухудшиться до $O(N^2)$. В таком случае лучше воспользоваться пирамидальной сортировкой. Быстрая сортировка также подвержена коварным ошибкам при неточной реализации. При незначительных ошибках в программе она плохо работает с определенными конфигурациями данных, и эти ситуации крайне трудно выявить.

В таблице 15.3 приведена сводка времени выполнения разных алгоритмов сортировки. В столбце «Оценка» оцениваются незначительные различия в скорости среди алгоритмов с одинаковой средней сложностью в O-синтаксисе. (Для сортировки Шелла оценка отсутствует из-за отсутствия других алгоритмов с той же скоростью.)

Таблица 15.3. Сравнение алгоритмов сортировки

Алгоритм	Средняя сложность	Сложность в худшем случае	Оценка	Дополнительные затраты памяти
Пузырьковая сортировка	$O(N^2)$	$O(N^2)$	Плохо	Нет
Сортировка методом выбора	$O(N^2)$	$O(N^2)$	Удовлетворительно	Нет
Сортировка методом вставки	$O(N^2)$	$O(N^2)$	Хорошо	Нет
Сортировка Шелла	$O(N^{3/2})$	$O(N^{3/2})$	–	Нет
Быстрая сортировка	$O(N*\log N)$	$O(N^2)$	Хорошо	Нет
Сортировка слиянием	$O(N*\log N)$	$O(N*\log N)$	Удовлетворительно	Да
Пирамидальная сортировка	$O(N*\log N)$	$O(N*\log N)$	Удовлетворительно	Нет

Графы

Графы занимают особое место в пантеоне структур данных. Они не используются для хранения данных общего назначения и не играют роль вспомогательных инструментов в других алгоритмах. Обычно графы применяются при непосред-

ственном моделировании реальных ситуаций. Структура графа отражает структуру задачи.

Если задача требует применения графа, никакая другая структура данных вам не подойдет, так что принимать решения по выбору структуры данных не нужно. Основное решение связано со способом представления графа: матрица смежности или списки смежности. Выбор зависит от того, является ли граф насыщенным (лучше использовать матрицу смежности) или разреженным (списки смежности).

С представлением в виде матрицы смежности алгоритмы обхода в глубину и в ширину выполняются за время $O(V^2)$, где V — количество вершин. Для представления в виде списка смежности они выполняются за время $O(V + E)$, где E — количество ребер. Алгоритмы построения минимального охватывающего дерева и выбора кратчайшего пути выполняются за время $O(V^2)$ при использовании матрицы смежности и за время $O((E + V)\log V)$ при использовании списков смежности. Оцените значения V и E для своего графа и вычислите, какое представление более уместно.

Внешнее хранение данных

В предыдущей части этой главы предполагалось, что данные хранятся в оперативной памяти. Но если объем данных слишком велик для хранения в оперативной памяти, то их приходится хранить на внешнем носителе — обычно в дисковых файлах. Внешнее хранение данных рассматривается во второй части главы 10, «Деревья 2-3-4» и главы 11, «Хеш-таблицы».

Будем считать, что данные сохраняются в файле блоками фиксированного размера и каждый блок вмещает несколько записей. (Запись в дисковом файле содержит те же данные, что и объект в оперативной памяти.) Запись, как и объект, обладает ключом, по которому производится выборка.

Также предполагается, что операции чтения и записи всегда выполняются с одним блоком, причем эти операции занимают значительно больше времени, чем любая обработка данных в оперативной памяти. Таким образом, для быстрой работы программы количество обращений к диску необходимо свести к минимуму.

Последовательное хранение

В простейшем варианте записи сохраняются в случайном порядке, а при поиске записи с заданным ключом выполняется их последовательный перебор. Новые записи просто вставляются в конец файла. Удаленные записи либо помечаются как удаленные, либо (по аналогии с массивом) другие записи сдвигаются для заполнения пробела.

В среднем операции поиска и удаления требуют чтения половины всех блоков, поэтому операции с последовательными данными не отличаются быстротой — они выполняются за время $O(N)$. Тем не менее при небольшом количестве записей этот вариант может быть приемлемым.

Индексированные файлы

Работа с внешними данными заметно ускоряется при использовании индексации. В этой схеме в оперативной памяти хранится индекс, связывающий пары из значения ключа и соответствующего номера блока. Чтобы обратиться к записи с заданным ключом, программа обращается к индексу и получает номер блока, после чего остается прочитать всего один блок за время $O(1)$.

Для набора данных могут использоваться сразу несколько индексов с разными ключами (один для фамилий, другой для номеров социального страхования и т. д.). Схема хорошо работает, пока индекс не станет слишком большим для хранения в памяти.

Как правило, сами индексы тоже хранятся в файлах на диске и загружаются в оперативную память по мере надобности.

Недостаток индексирования заключается в том, что индекс в какой-то момент необходимо построить. Скорее всего, для этого придется последовательно прочитать все содержимое файла, поэтому создание индекса выполняется медленно. Кроме того, индекс должен обновляться при добавлении новых элементов в файл.

В-деревья

В-деревья представляют собой многопутевые деревья, часто используемые для внешнего хранения данных; их узлы соответствуют блокам на диске. Как и в других разновидностях деревьев, алгоритмы последовательно проходят сверху вниз по дереву, читая по одному блоку на каждом уровне. В-деревья обеспечивают поиск, вставку и удаление записей за время $O(\log N)$ — операции выполняются довольно быстро даже для очень больших файлов. С другой стороны, программирование В-деревьев является нетривиальной задачей.

Хеширование

Если на внешнем носителе можно использовать вдвое больше пространства, чем обычно занимает файл, то внешнее хеширование может оказаться хорошим вариантом. Выборка данных выполняется за такое же время, как при индексировании ($O(1)$), но хеширование позволяет работать с файлами большего размера.

На рис. 15.2 представлен упрощенный алгоритм выбора структуры для внешнего хранения данных.

Виртуальная память

Иногда проблемы с доступом к диску решаются поддержкой виртуальной памяти на уровне операционной системы без особых усилий с вашей стороны. Если программа читает файл, который не помещается в оперативной памяти, то подсистема виртуальной памяти читает небольшую часть файла и сохраняет остальное на

диске. Когда программа обращается к другим частям файла, данные автоматически читаются с диска и размещаются в памяти.

Программист применяет алгоритмы ко всему файлу так, словно все данные находятся в памяти одновременно, а операционная система сама прочитает нужную часть файла, если она еще не была загружена в память.

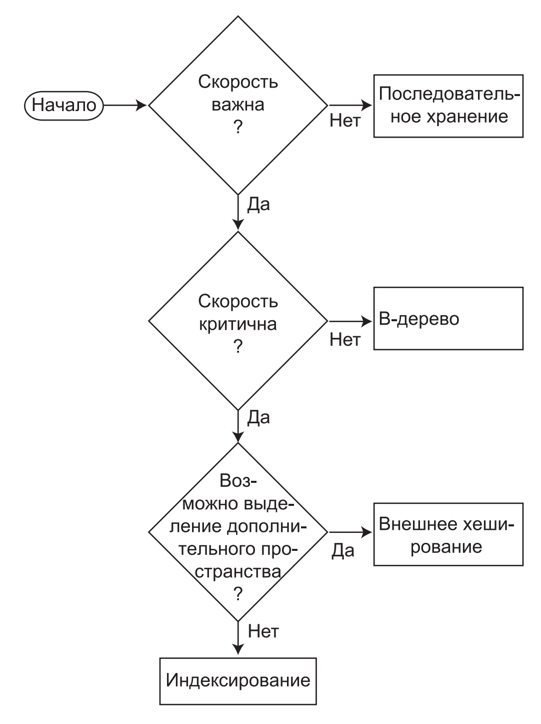


Рис. 15.2. Выбор структуры данных для внешнего хранения

Конечно, программа будет работать намного медленнее, чем если бы весь файл находился в памяти, однако замедление неизбежно и при обработке файла блок за блоком одним из алгоритмов внешнего хранения данных. Итак, в некоторых ситуациях можно просто попытаться проигнорировать тот факт, что файл не помещается в памяти, и посмотреть, как исходные алгоритмы будут работать при использовании виртуальной памяти. Такое решение особенно хорошо подходит для файлов, размер которых ненамного превышает объем доступной памяти.

Итоги

Наше изучение структур данных и алгоритмов подошло к концу. Это чрезвычайно обширная и сложная тема, и стать экспертом после чтения одной книги невозможно, но мы надеемся, что книга помогла вам изучить основы. В приложении Б приведен список книг для дальнейшего изучения.

Приложение A

Приложения Workshop и примеры программ

В этом приложении рассматриваются некоторые подробности запуска приложений Workshop и примеров программ.

- Графические программы Workshop демонстрируют строение и принципы работы деревьев и других структур данных.
- Примеры программ, код которых приводится в тексте книги, содержат полностью работоспособный код Java.

Мы также рассмотрим пакет Sun Microsystems Java 2 Standard Edition (J2SE) Software Development Kit (SDK), который может использоваться не только для запуска приложений и программ из книги, но и для их изменения и написания собственных программ.

Приложения и примеры можно найти на странице этой книги на веб-сайте издательства «Питер» (*www.piter.com*) и на сайте Sams по адресу *www.samspublishing.com*.

Приложения Workshop

Апплет (applet) — особая разновидность Java-программ, код которых легко передается по Интернету. Так как апплеты ориентированы на работу в интернет-среде, они могут запускаться на любой компьютерной платформе с подходящим браузером или другой программой.

В этой книге приложения Workshop обеспечивают динамическую, интерактивную демонстрацию концепций, изложенных в тексте. Например, в главе 8, «Двоичные деревья», представлено приложение Workshop, в рабочей области которого отображается дерево. При помощи кнопок приложения пользователь может проследить за процессом вставки нового узла в дерево, удаления существующего узла, обхода дерева и т. д. В других главах тоже используются соответствующие приложения Workshop.

Приложения Workshop можно запустить сразу же после загрузки в большинстве основных браузеров, включая Microsoft Internet Explorer и Netscape Communicator. В коммерческих пакетах Java-разработки также имеются свои программы для просмотра аплетов. В частности, для запуска приложений Workshop можно воспользоваться программой *appletviewer* из пакета SDK.

Чтобы запустить приложение Workshop в типичном браузере, выполните команду Файл ▸ Открыть и откройте соответствующий каталог. Каждое приложение Workshop состоит из подкаталога, в котором находятся файлы с расширением *.class*, и одного файла с расширением *.html*. Откройте файл *.html*. В окне браузера должна открыться рабочая область приложения.

Примеры программ

В примерах программного кода мы стараемся по возможности просто показать, как структуры и алгоритмы, описанные в книге, могут быть реализованы на языке Java. Примеры кода не предназначены для загрузки по Интернету, они просто запускаются как обычные программы на конкретном компьютере.

Программы Java могут выполняться либо в консольном, либо в графическом режиме. Для простоты наши примеры работают в консольном режиме; это означает, что результаты выводятся в текстовом виде, а входные данные вводятся с клавиатуры. В среде Windows программы консольного режима выполняются в окне MS-DOS. В консольном режиме графика не поддерживается.

Код примеров приведен в тексте книги. Исходные файлы, содержащие тот же код, можно загрузить с сайта издательства.

Sun Microsystems SDK

Для выполнения как приложений Workshop, так и примеров можно использовать служебные программы, входящие в Sun SDK. Пакет SDK загружается с веб-сайта Sun по адресу *www.sun.com* (найдите ссылку для загрузки Java 2 Standard Edition (J2SE) SDK). Пакет довольно большой, однако в нем есть все необходимое не только для запуска приложений и примеров из книги, но и для самостоятельного Java-программирования.

Программы командной строки

SDK работает в текстовом режиме, а его различные программы запускаются из командной строки. В системе Windows для доступа к командной строке необходимо открыть окно MS-DOS. Щелкните на кнопке `Пуск` и найдите в меню пункт `Командная строка`. Обычно эта команда находится в группе `Стандартные`.

Введите в приглашении MS-DOS команду *cd* для перехода в подкаталог жесткого диска, где хранится приложение Workshop или программа-пример. Запустите приложение или программу при помощи соответствующей утилиты SDK, как описано ниже.

Настройка пути

В системе Windows местонахождение программ SDK можно указать в директиве PATH в файле *autoexec.bat*. Это делается для того, чтобы их было удобно запускать из любого подкаталога. Директива PATH может быть автоматически включена в файл *autoexec.bat* во время установки SDK. Если это не было сделано, запустите программу Блокнот и вставьте строку

```
SET PATH=C:\JDK1.4.0\BIN;
```

в файл *autoexec.bat* после всех остальных директив SET PATH. Файл *autoexec.bat* размещается в корневом каталоге. Чтобы активизировать изменения в системном пути, закройте окно командной строки и откройте новое. (При необходимости укажите другой номер версии и имя каталога.)

Запуск приложений Workshop

Чтобы запустить приложения Workshop средствами SDK, сначала перейдите в нужный каталог командой *cd* в окне MS-DOS. Например, чтобы запустить приложение Array Workshop из главы 2, «Массивы», выполните следующие команды:

```
C:\>cd javaapps
C:\javaapps>cd chap02
C:\javaapps\chap02>cd Array
```

Затем выполните файл *.html* в программе *appletviewer* из SDK:

```
C:\javaapps\chap02\Array>appletviewer Array.html
```

Иногда загрузка требует времени, будьте терпеливы. Внешний вид рабочей области приложения должен быть близок к иллюстрациям, приведенным в тексте. Точного совпадения не будет, потому что разные браузеры и программы по-разному интерпретируют HTML и Java.

Как было сказано ранее, для запуска приложений Workshop также можно использовать все основные браузеры.

Работа с приложениями Workshop

В каждой главе приводятся инструкции по работе с конкретными приложениями Workshop. Обычно для выполнения каждой операции необходимо многократно нажимать ту или иную кнопку. Например, каждое нажатие кнопки Ins в приложении Array Workshop выполняет один шаг процесса вставки. Как правило, в приложении выводится сообщение с описанием того, что происходит на каждом шаге.

Всегда завершайте каждую операцию, то есть каждую серию нажатий кнопки, прежде чем начинать другую операцию при помощи другой кнопки. Например, продолжайте нажимать кнопку Find, пока не будет найден элемент с заданным ключом, а в приложении появится надпись *Press any button*. Только после этого следует переходить к операции с другой кнопкой, например вставке нового элемента кнопкой Ins.

В приложениях из главы 3, «Простая сортировка», и главы 7, «Нетривиальная сортировка», присутствует кнопка `Step` для пошагового выполнения сортировки. Кнопка `Run` запускает сортировку на высокой скорости без дополнительных нажатий кнопок. Щелкните на кнопке `Run` и проследите, как столбцы выстраиваются в упорядоченную последовательность. Чтобы приостановить сортировку, нажмите кнопку `Step`, а чтобы продолжить — нажмите `Run`.

Код приложений Workshop не предназначен для анализа, так как большая его часть относится к графическому выводу. Соответственно листинги этих приложений не приводятся.

Запуск примеров

Каждая программа-пример состоит из подкаталога с файлом *.java* и нескольких файлов с расширением *.class*. Файл *.java* содержит исходный код, который необходимо откомпилировать перед запуском. Файлы *.class* уже откомпилированы и готовы к запуску в интерпретаторе Java.

Для запуска примеров прямо из файлов *.class* можно воспользоваться интерпретатором Java из Sun SDK. Для каждой программы имя одного из файлов *.class* завершается буквами *App*; именно этот файл должен запускаться интерпретатором.

В командной строке перейдите в соответствующий каталог (командой *cd*) и найдите файл с суффиксом *App*. Например, для программы `insertSort` (глава 3) откройте подкаталог *InsertSort* главы 3 (не перепутайте каталог приложений Workshop с каталогом примеров!). Вы найдете в нем файл *.java* и несколько файлов *.class*, в том числе *insertSortApp.class*. Программа запускается командой

```
C:\chap03\InsertSort>java insertSortApp
```

Не вводите расширение после имени файла. Программа `insertSort` выводит данные до и после сортировки в текстовом виде. В некоторых примерах отображается приглашение для ввода команд с клавиатуры.

Компилирование примеров

Поэкспериментируйте с примерами — измените их код, откомпилируйте и запустите измененную новую версию. Также можно написать собственное приложение «с нуля», откомпилировать и запустить его. Java-код компилируется программой `javac`, которой передается имя *.java*-файла. Например, чтобы откомпилировать программу `insertSort`, перейдите в каталог *insertSort* и введите команду

```
C:\chap03\insertSort>javac insertSort.java
```

На этот раз расширение *.java* должно быть включено в командную строку. Команда компилирует файл *.java* на несколько файлов *.class* по количеству классов в программе. Если исходный код содержит ошибки, они отображаются на экране.

Редактирование исходного кода

Существует много разных редакторов для изменения и написания исходных файлов *.java*. Например, можно запустить редактор для MS-DOS *edit* в окне командной строки или же воспользоваться редактором Блокнот для Windows. Также имеется много разных коммерческих редакторов.

Не используйте программы форматирования текста (например, Microsoft Word) для редактирования исходных файлов. Как правило, такие программы включают в свои файлы служебные символы и информацию о форматировании, которую интерпретатор Java не поймет.

Завершение примеров

Выполнение любой консольной программы можно прервать нажатием клавиш Ctrl+C. В некоторых примерах предусмотрена специальная команда завершения, упоминаемая в тексте (например, нажатие Enter в начале строки), но другие программы приходится завершать нажатием Ctrl+C.

Одноименные файлы классов

Довольно часто в нескольких приложениях Workshop или примерах программ используются файлы *.class* с одинаковыми именами. Учтите, что эти файлы могут и не быть идентичными. Приложение Workshop или пример может работать некорректно с «чужими» файлами классов, даже если эти файлы имеют подходящие имена.

С выбором запускаемого файла проблем быть не должно, так как все файлы одной программы хранятся в одном подкаталоге. Но если вы перемещаете файлы вручную, следите за тем, чтобы не скопировать файл в другой каталог. Это может создать проблемы, которые будет трудно обнаружить.

Другие системы разработки

Кроме Sun SDK, существует много других систем разработки на Java от Symantec, Microsoft, Borland и т. д. Компания Sun также предлагает систему разработки на Java, которая называется Sun ONE Studio 4 (ранее она называлась Forte). Обычно эти продукты работают быстрее и более удобны в работе, чем SDK. Все функции — редактирование, компилирование и исполнение — объединяются в одном окне.

Чтобы эти системы разработки могли использоваться с примерами программ, приведенными в книге, они должны поддерживать Java версии 1.4.0 и выше. Многие примеры программ (прежде всего, те, в которых используется ввод данных пользователем) не будут компилироваться продуктами, рассчитанными на более ранние версии Java. (Впрочем, после внесения небольших изменений, некоторые из них упомянуты в главе 1, «Общие сведения», можно заставить файлы *.java* компилироваться в более старых системах разработки.)

Приложение Б
Литература

В этой главе упоминаются книги, посвященные различным аспектам разработки ПО, в том числе и структурам данным с алгоритмами. Список не претендует на объективность; на эти темы написано много других замечательных книг.

Структуры данных и алгоритмы

Наиболее авторитетным источником по структурам данных и алгоритмам является книга Дональда Кнута (Donald E. Knuth) «The Art of Computer Programming» (издательство Addison Wesley, 1998). Этот фундаментальный труд впервые был опубликован в 1970-х годах, а сейчас вышло уже третье издание. Книга состоит из трех томов: «Основные алгоритмы», «Получисленные алгоритмы» и «Сортировка и поиск». Том 3 наиболее тесно связан с темой нашей книги. Материал рассчитан на читателя с хорошей математической подготовкой и не является «легким чтивом», однако книга считается «Библией» для всех, кто серьезно изучает данную область.

Чуть более доступно написана книга Роберта Седжвика (Robert Sedgewick) «Algorithms in C++» (издательство Addison Wesley, 1998). Книга была написана на основе материала более ранней книги «Algorithms» (издательство Addison Wesley, 1988), в которой примеры кода были написаны на Pascal. Это полный и авторитетный источник информации. Стиль изложения (и текст, и примеры) весьма лаконичен, поэтому книгу необходимо читать внимательно и вдумчиво.

В книге «Algorithms in Java» Роберта Седжвика (Robert Sedgewick) и Майкла Шидловски (Michael Schidlowsky) (издательство Addison Wesley, 2002) примеры написаны на языке Java.

Хороший учебник в области структур данных и алгоритмов для студентов — книга Дженет Причард (Janet J. Prichard) и Фрэнка Каррано (Frank M. Carrano) «Data Abstraction and Problem Solving with C++: Walls and Mirrors» (издательство Benjamin Cummings, 2001). В книге много иллюстраций, а каждая глава завершается упражнениями и проектами для самостоятельной работы. Версия для языка Java называется «Data Abstraction and Problem Solving with Java: Walls and Mirrors».

В книге Брайана Флемига (Bryan Flamig) «Practical Algorithms in C++» (издательство John Wiley and Sons, 1995) к многим стандартным темам добавляются другие, редко встречающиеся в других книгах, такие как генераторы алгоритмов и строковый поиск.

Книга Джона Луиса Бентли (Jon Louis Bentley) «Programming Pearls» (издательство Addison Wesley, 1999) была написана в 1986 году, но содержит массу по-

лезнейшей информации для программистов. Большая часть материала относится к структурам данных и алгоритмам.

Также в области структур данных и алгоритмов заслуживают внимания книги Тимоти Бадда (Timothy A. Budd) «Classic Data Structures in C++» (издательство Addison Wesley, 2000), Марка Аллена Вайсса (Mark Allen Weiss) «Data Structures and Problem Solving Using C++» (издательство Addison Wesley, 1999) и Йедидья Лангсэма (Yedidyah Langsam) и др. «Data Structures Using C and C++» (издательство Prentice Hall, 1996).

Объектно-ориентированные языки программирования

Доступное и основательное введение в Java и объектно-ориентированное программирование приведено в книге Стивена Гилберта (Stephen Gilbert) и Билла Маккарти (Bill McCarty) «Object-Oriented Programming in Java» (издательство Waite Group Press, 1997).

Если вас интересует язык C++, попробуйте книгу Роберта Лафоре «Object-Oriented Programming in C++, Fourth Edition» (Объектно-ориентированное программирование в C++. СПб.: Питер, 2011).

Книга Кена Арнольда (Ken Arnold), Джеймса Гослинга (James Gosling) и Дэвида Холмса (David Holmes) «Java Programming Language, Third Edition» (издательство Addison Wesley, 2000) посвящена синтаксису Java. Безусловно, она авторитетна (хотя и более лаконична по сравнению с другими книгами): Гослинг, работающий в Sun Microsystems, является создателем языка Java.

В многотомной серии книг Кей Хорстман (Cay S. Horstmann) и Гэри Корнелла (Gary Cornell) «Core Java 2, Fifth Edition» (издательство Prentice Hall, 2000) глубоко, но очень доступно изложено почти все, что необходимо знать о программировании на Java.

Объектно-ориентированное проектирование и разработка

Простое и доступное введение в тему разработки ПО представлено в книге Скотта Эмблера (Scott W. Ambler) «The Object Primer: The Application Developer's Guide to Object-Orientation, Second Edition» (издательство Cambridge University Press, 2001). Эта небольшая книга простым языком объясняет, как происходит проектирование крупного программного проекта. Название не совсем точно — книга выходит далеко за рамки простых объектно-ориентированных концепций.

Книга Стивена Гилберта и Билла Маккарти «Object-Oriented Design in Java» (издательство Waite Group Press, 1998) отличается доступностью изложения.

Работа Грэди Буча (Grady Booch) «Object-Oriented Analysis and Design with Applications» (издательство Addison Wesley, 1994) считается классикой в области объектно-ориентированного проектирования. Автор — один из первопроходцев в этой области, создатель схемы Буча для обозначения отношений между классами. Пожалуй, начинающему эта книга покажется сложной, но для более опытных читателей она будет бесценной.

Одна из ранних работ в области объектно-ориентированного проектирования, книга Фредерика Брукса (Frederick P. Brooks, Jr.) «The Mythical Man-Month» (издательство Addison Wesley, 1975, переиздана в 1995), предельно четко и осмысленно объясняет необходимость качественного проектирования программных продуктов. Говорят, по количеству проданных экземпляров эта книга опережает любую другую книгу компьютерной тематики.

В числе других хороших работ, посвященных объектно-ориентированному проектированию, заслуживают упоминания книга Тимоти Бадда (Timothy Budd) «An Introduction to Object-Oriented Programming, Third Edition» (издательство Addison Wesley, 2002); книга Артура Рила (Arthur J. Riel) «Object-Oriented Design Heuristics» (издательство Addison Wesley, 1996); и книга Эрика Гамма (Erich Gamma) и др. «Design Patterns: Elements of Reusable Object-Oriented Software» (Приемы объектно-ориентированного проектирования, СПб.: Питер, 2011).

Приложение В
Ответы на вопросы

Глава 1. «Общие сведения»

1) вставки, поиска, удаления
2) сортировкой.
3) c
4) ключом
5) b
6) a
7) d
8) метод
9) «точка»
10) типами данных

Глава 2. «Массивы»

1) d
2) да
3) b
4) нет
5) new
6) d
7) интерфейс
8) d
9) возведению в степень
10) 3
11) b
12) 6
13) нет
14) a

15) постоянное
16) объекты

Глава 3. «Простая сортировка»

1) d
2) сравнение, сортировка, копирование
3) нет
4) a
5) нет
6) b
7) нет
8) три
9) сортируются элементы с индексами, меньшими либо равными `outer`.
10) c
11) d
12) копирования
13) b
14) элементы с индексами, меньшими `outer`, частично отсортированы.
15) b

Глава 4. «Стеки и очереди»

1) 10
2) b
3) Last In, First Out и First In, First Out
4) нет, наоборот
5) b
6) он вообще не перемещается
7) 45
8) нет, за время $O(1)$.
9) c
10) $O(N)$
11) c
12) да

13) b
14) да, понадобится метод для определения наименьшего значения
15) a

Глава 5. «Связанные списки»

1) b
2) первый
3) d
4) 2
5) 1
6) c
7) current.next=null;
8) он уничтожается уборщиком мусора Java.
9) a
10) пустой
11) в связанном списке.
12) одно, если в элементах хранится ссылка на предыдущий элемент.
13) двусторонний список.
14) b
15) Вероятно, список. В обоих случаях операции `push()` и `pop()` выполняются за время $O(1)$, но список более эффективно использует память.

Глава 6. «Рекурсия»

1) 10
2) d
3) 2
4) 10
5) нет
6) `"ed"`
7) b
8) c
9) последовательного разделения
10) диапазон ячеек для поиска

11) количество перемещаемых дисков
12) c
13) b
14) b
15) стек

Глава 7. «Нетривиальная сортировка»

1) c
2) 40
3) d
4) нет
5) $O(N*\log N)$, $O(N^2)$
6) a
7) опорным
8) d
9) да
10) c
11) разбиением полученных подмассивов
12) b
13) опорного элемента
14) $\log_2 N$
15) true

Глава 8. «Двоичные деревья»

1) $O(\log N)$
2) b
3) да
4) 5
5) c
6) узел, дерево
7) a
8) c
9) поиск

10) А, узлов левого поддерева А
11) d
12) 2*n + 1
13) нет
14) сжатия
15) c

Глава 9. «Красно-черные деревья»

1) упорядоченных (в прямом или обратном порядке) данных
2) b
3) нет
4) d
5) b
6) поворот, изменение цвета узлов
7) красный
8) a
9) левым потомком, правым потомком
10) d
11) узла, его двух потомков
12) b
13) да
14) a
15) да

Глава 10. «Деревья 2-3-4»

1) b
2) сбалансировано
3) 2
4) нет
5) b
6) разбиении корневого узла
7) a
8) 2

9) переключению цветов
10) b
11) $O(\log N)$
12) d
13) много
14) да
15) a

Глава 11. «Хеш-таблицы»

1) $O(1)$
2) Хеш-функция
3) d
4) линейным пробированием
5) 1, 4, 9, 16, 25
6) b
7) связанного списка
8) 1.0
9) да
10) d
11) размеру массива
12) нет
13) a
14) нет
15) одном блоке

Глава 12. «Пирамиды»

1) b
2) ключи как правого, так и левого потомка меньше либо равны ключу родителя.
3) корня
4) a
5) a
6) c
7) массив (или связанный список)

8) вверх
9) b
10) один

Глава 13. «Графы»

1) ребро, вершины
2) подсчитать количество единиц и разделить на 2 (предполагается, что диагональ матрицы заполнена нулями).
3) узел
4) d
5) A:B, B:A—>C—>D, C:B, D:B
6) a
7) 3
8) c
9) деревом
10) нет
11) да
12) d
13) нет (по определению)
14) вершины еще остаются, но ни одна вершина не имеет преемников.

Глава 14. «Взвешенные графы»

1) ребер
2) d
3) нет
4) ребро с наименьшим весом
5) b
6) уже является конечной для ребра с меньшим весом
7) нет
8) a
9) да
10) a
11) b

12) алгоритм Уоршелла
13) матрицы смежности
14) $2N$, где N — количество полей на доске минус 1
15) нет

Об авторе

Роберт Лафоре имеет ученые степени в области электротехники и математики. Он работал системным аналитиком в лаборатории Лоуренса в Беркли, учредил собственную компанию, занимающуюся разработкой ПО, а также написал ряд популярных книг о программировании, в том числе «C++ Interactive Course» и «Object-Oriented Programming in C++» (Объектно-ориентированное программирование в С++. СПб.: Питер, 2011). Среди его более ранних работ широкую известность получили книги «Assembly Language Primer for the IBM PC and XT» и (на заре компьютерной революции) «Soul of CP/M».

Алфавитный указатель

Г

Д

З

И

К

Л

М

Н

О

П

Т

У

Ф

Х

Ц

Ш

Э

Роберт Лафоре

Структуры данных и алгоритмы в Java.
Классика Computers Science.
2-е издание

Серия «Классика computer science»

Перевел с английского Е. Матвеев

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Юрченко*
Ведущий редактор	*Ю. Сергиенко*
Художественный редактор	*К. Радзевич*
Корректор	*В. Листова*
Верстка	*Е. Егорова*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано в печать 26.06.13. Формат 70х100/16. Усл. п. л. 56,760. Тираж 1000. Заказ
Отпечатано в полном соответствии с качеством предоставленных издательством материалов
в Чеховский Печатный Двор. 142300, Чехов, Московская область, г. Чехов, ул. Полиграфистов, д. 1.